(56)

The NetLog: An Efficient, Highly Available, Stable Storage Abstraction

by

Arvind Parthasarathi

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1998

Author .....
Department of Electrical Engineering and Computer Science
April 17, 1998

Certified by .....
Barbara Liskov
Ford Professor of Engineering
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Committee on Graduate Students

The NetLog: An Efficient, Highly Available, Stable Storage Abstraction

by

Arvind Parthasarathi

Submitted to the Department of Electrical Engineering and Computer Science
on April 17, 1998, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

In this thesis I present the NetLog, an efficient, highly available, stable storage abstraction for distributed computer systems. I implemented the NetLog and used it in the Thor distributed, object-oriented database. I also show that the NetLog outperforms the most commonly used alternative on both present and future hardware configurations.

The NetLog defines a log abstraction that is more suited to application requirements than other proposed log abstractions. Unlike traditional storage mediums, such as magnetic disks, that provide only reliability, the NetLog provides both reliable and available storage through the use of primary copy replication. By hiding all details of processing from applications and implementing a generic interface, the NetLog can be used by any application and applications that accessed reliable storage now get the benefits of availability with little code change.

Within Thor, I developed implementations of the NetLog that survive one and two server failures. I also implemented a disk based log so that the performance of the two logs could be compared within a real system. I developed a model of both logs to extend the comparison to other system and technology configurations. My performance results, from experiments and modeling, show that the availability provided by the NetLog comes for free as the NetLog always performs faster than the disk log, and performs an order of magnitude faster for small data sizes.

Thesis Supervisor: Barbara Liskov
Title: Ford Professor of Engineering

Acknowledgments

I would like to thank my advisor, Barbara Liskov, for her help and guidance. Her advice throughout this work has been invaluable. Her infinite patience in reading countless drafts of this thesis improved it immeasureably.

Miguel Castro showed me the error of my ways in measuring flush times in Thor and his comments on the development and presentation of the model were very helpful. Atul Adya played the role of resident Thor guru and helped me solve any problems that I had working with the system.

I would like to thank Atul Adya, Kavita Bala, Phil Bogle, Chandra Boyapati, Miguel Castro, Satyan Coorg, Jason Hunter, Umesh Maheswari, Andrew Myers, and Doug Wyatt for the numerous technical and philosophical discussions that were an inseparable part of the graduate school experience.

Finally, I would like to thank my parents, who were always there for me and without whose love, support and encouragement, none of this would have been possible. And my brother who insisted that he be acknowledged or else ...

Contents

List of Figures

List of Tables

Chapter 1

Introduction

An important goal for computer systems that provide storage for objects on the Internet is to make that storage both reliable and available. Reliability is attained by ensuring that the information in these objects is not lost or corrupted on system failure including system crashes and media failures. Reliability also guarantees that the effects of an operation executed on a data object are visible to all those that access the data object after the operation has completed regardless of any failures that might take place. Availability is attained by ensuring that, in addition to reliability, the objects are always accessible when needed, even in the face of system and media failures and network errors and partitions.

Reliability is provided through the use of *stable storage*. Stable storage is storage that is both persistent and failure atomic. A device/storage is said to be failure atomic when a write to the device/storage is guaranteed to either take place completely or not at all. If a failure occurs during a write and the write is unable to succeed, the state of the storage will be restored to what it was before the write began. The exact nature and type of data that is kept in stable storage and how the system uses it varies from system to system. Stable storage has primarily been used in transactional systems to provide consistency but increasingly is of interest in non-transactional systems, like file systems and network servers, to recover from failures [SMK93, CD95].

Most systems implement stable storage on magnetic disks [G78, BJW87, H87, BEM90, KLA90]. Disks store information through magnetic recording and retain information even in the face of system crash and power failure. If stable storage is implemented on a single disk, then the storage does not survive media failure. In order for disk based stable storage to survive media failures, systems have used mirrored disks where the data is written to two disks so that if one disk fails the other is still accessible.

Using disks to implement stable storage suffers from two major shortcomings. First, writing to disk is time consuming and the time taken to write to disk has been increasing relative to other system latencies. Second, and more important, is that the stable storage provided by disks is not available; when the disk unit is down (say due to power failure), the data becomes unavailable.

This thesis defines a new abstraction, the NetLog, that provides efficient, highly available stable storage. The thesis provides an implementation of this abstraction within a particular

object storage system (the Thor distributed database) and also provides a performance analysis of the NetLog abstraction on both present and future hardware. The NetLog derives its name from tha fact that it uses a *log* of the data at systems connected by a *net*work to provide reliable and available storage for data.

The rest of this chapter is organized as follows. I first describe the client-server system model, which is the focus of this thesis. I then go on to introduce replication techniques and protocols that are used to achieve high availability in client-server distributed systems. I then discuss the contributions of this thesis, and conclude this chapter with a brief outline of the rest of the thesis.

1.1 The Client-Server System

In a *client-server* system, the machine on which the application is run (the client) is different from the machine on which the program that services the application runs (the server). The basic idea behind client-server systems is to separate the application from the service it requires to perform its task. A variety of systems use the client-server model; file systems, distributed databases, registry servers, and network servers, to name a few. For example, in a file system, the files are stored at the server and user applications run on clients that access files from the servers. A client reads a file by bringing the file to the client machine and when the client has finished using the file, it is written back to the server. In the example of a network time server, clients query the server to get the current system time at the server. As can be seen from the above examples, the exact operations performed by the clients at the servers varies from system to system but the notion of clients performing operations on (or getting service from) the servers remains.

This thesis focusses on client-server based distributed systems because continuous service for clients can be guaranteed. By separating the clients and servers, one server can replace another server which has failed without the client application being affected. Further, the servers can be made more robust (by equipping them with a UPS) and more secure (by placing them in secure places and running advanced security checks) without having to undergo these costs for the clients. Also, placing the program that services applications at the server facilitates sharing of the service between multiple clients and placing the applications on separate client machines permits the system to scale and seamlessly handle a large number of applications.

1.2 Replication Techniques

The concept of using replication to provide fault tolerance to mask component failures was first proposed by von Neumann [V56]. A good background and description of replication techniques is given in [BHG87]. The main benefits of replication are:

- To increase reliability by having multiple independent copies of the data object. If one server goes down, perhaps even permanently, data is not lost.

- To increase the availability of the data by ensuring that the data object is accessible even if one server goes down. A server crash should not bring down the entire system until the server can be rebooted.

- To improve performance of the system. The workload can be split over multiple servers so that a single server is not a bottleneck. By having multiple servers, the least loaded server can be used.

The main issue in replication is transparency to the user of the application that runs on the client. At one end, users could not only be aware of the replication process but also control it. At the other end of the continuum are systems where the users are unaware of replication. In a user-initiated replication scheme, the user places objects on different servers and is responsible for ensuring and interpreting the consistency between the copies. In a completely transparent replication scheme, the user is not aware of the replication but can see the effects in terms of increased availability. In a transparent replication scheme, operations are automatically performed on enough copies of the object to ensure that the effects of the operation survive all subsequent failures. Polyzois and Garcia-Molina [PG92] evaluate replication schemes based on the level of user-transparency and replica consistency in client server based transaction processing systems.

For transparent replication, the key issue is ensuring that the copies of the data at the different replicas are consistent. Sending a message to every copy in sequence will not work as the process sending the messages might crash before it is done. There are two main approaches to solving this problem, namely the primary copy approach, and the quorum consensus approach. In both cases, to survive f server failures, the number of servers in the system, N, must be at least $2f + 1$ [L96].

1.2.1 Primary Copy

In the primary copy replication scheme [AD76, O88, T79, G83], one server is designated as a primary and the rest are designated as backups. Client operations are directed only to the primary. When the primary receives a message describing an operation from a client, it performs the operation locally and writes the results to disk. The primary then sends the operation to the backups so that they can perform it as well. After the operation has been performed at a backup, a backup writes the results to disk and sends an acknowledgement to the primary. In order to ensure that the effects of the operation survive all subsequent failures, the primary must receive acknowledgements from enough backups (f backups to survive f failures) before it tells the client that the operation has been carried out successfully.

If the backup or primary crashes, the servers of the group that are still up reorganize themselves into another configuration containing one server as the primary and the rest as backups. After this reorganization, the system can continue to process client operations. The protocol that is used to recover in the face of server failures is called a *view change* [G83, ESC85, ET86, O88]. Replication protocols that not only survive server failure but also network partitions are discussed in [DGS85].

1.2.2 Quorum Consensus

While the primary copy scheme discussed above was not concerned about the type of client operation, the quorum consensus scheme is more limited. It classifies client operations into either a read or a write of the data. Writes modify the state of the data and must be conveyed to enough replicas to ensure that the effects of the write are reliable.

The quorum consensus scheme uses voting [G79, ES83, P86] where clients request and acquire the permission of multiple replicas before reading or writing the data object. The number of servers, N_r, whose assent is required for a read operation is called the *read quorum* and the number of servers, N_w, whose assent is required for a write operation is called the *write quorum.* Each server can be thought of as having a token that can be acquired by clients. When the client acquires enough tokens (to form the required quorum), it performs the read or write operation and returns the tokens to the servers. Servers may grant multiple read tokens but cannot issue a write token unless no tokens have been issued. And when a write token has been issued, no token can be issued. If the total number of servers is N, then to ensure that read and write operations by clients are consistent, it must hold that $N_r + N_w > N$.

An interesting problem that arises in setting read and write quorums is that read operations are much more common than writes and consequently, to make the common case fast, N_r is usually set small. In such a situation, N_w is close to N (in fact, typically $N_w = N$) and it may be impossible to acquire a write quorum if a few servers are down. This problem was fixed in [VT88] by using dummy servers called *ghosts* that take part only in write quorums and not in read quorums.

1.2.3 Primary Copy vs. Quorum Consensus

The primary copy approach is widely preferred to the quorum consensus approach for a number of reasons. Firstly, the primary copy scheme is simpler than the quorum protocol because it is easier for clients to interact with one server than exchange messages with multiple servers and acquire quorums. Secondly, in the quorum scheme, concurrent writes can cause deadlock and avoiding such situations introduces additional complexity into the protocol. Thirdly, the quorum scheme is further complicated as it needs to handle clients crashing after they have acquired tokens from servers. Finally, unlike the quorum consensus approach, the primary copy scheme does not classify client operations into reads and writes and thus is easily extendible.

A big problem with the quorum protocols is that they do not scale with number of clients. Every read and write will generate traffic between the client and most servers. Clients are generally connected to the servers through low speed connections and this traffic may overload their network. On the other hand in the primary copy scheme, there is only one round-trip message communication between the client and the primary. The communication between the primary and the backups is managed more easily as they are usually connected by a much higher speed connection and messages generated by multiple clients can be piggybacked. Studies [GHO96] have shown that as the number of clients and servers is increased, the network traffic in the quorum protocols is high and causes network

congestion and system slowdown.

1.3 Contributions of this Thesis

This section discusses the main contributions of this thesis.

1.3.1 Design of the NetLog abstraction

The NetLog is an efficient, highly available, stable storage abstraction that can be used by servers in any distributed system. The NetLog replicates data on a number of machines to achieve high availability by using the primary copy replication technique. In the primary copy scheme, the primary and backups record information on disk and carry out a protocol to ensure that they are synchronized and the data is consistent. The primary and the backups also carry out a view change protocol on failover. The details of all this activity is encapsulated within the NetLog abstraction and hidden from the servers.

The NetLog provides the server program with access to stable storage through a simple interface. This interface is identical to the traditional log abstraction used by many systems that implement stable storage on disk. Servers that use the traditional log abstraction to access reliable storage can now use the NetLog and achieve availability with little rewrite of server code.

One key optimization of the NetLog design is to assume that all servers are equipped with an Uninterruptible Power Supply (UPS). The UPS enables the NetLog to be stored in main memory at both the primary and the backups rather than on disk. Log data is not lost on power failure as the UPS affords time to write the log data to disk. By replacing disk I/O with one round-trip network communication, the NetLog surmounts the twin problems of availability and performance faced by disks: the NetLog design provides highly available stable storage and potentially leads to better system performance than disks.

1.3.2 Implementation of the NetLog in Thor

The NetLog abstraction has been implemented and used in the Thor distributed, object-oriented database [LAC96]. This implementation not only demonstrates the utility of the NetLog in a real system but also provides an environment for testing and studying the performance of the NetLog abstraction. Thor is based on the client-server paradigm and Thor servers use the stable storage provided by the NetLog to manage the database. Configurations of the NetLog that permit 1 and 2 server failures were developed. The system was tested extensively and its results were compared with previous versions of Thor for both consistency and correctness.

A disk log abstraction was also built into Thor. By making the disk log and the NetLog implement the same interface, Thor servers can use either the disk log or the NetLog in a seamless fashion.

1.3.3 Performance of the NetLog in Thor

As reported in [ONG93], log performance is significant in terms of overall system performance. In the NetLog design, the UPS allows the substitution of a round trip network communication for a disk write. There is therefore potential for better performance and this was demonstrated in the Harp system [LGG91]. Harp is a highly available, replicated file system that used an in-memory log with UPS support to remove synchronous disk operations. In [LGG91], the performance of the Harp system was compared to that of unreplicated NFS (which used synchronous disk writes) on standard file system benchmarks and significant performance improvement was reported. But since the Harp system had a specialized implementation, the results are not generalizeable beyond file systems. In this thesis, the performance of the NetLog is characterized in a manner that is easily applicable to any distributed system that uses a log, namely, the time to flush data to stable storage.

The performance of the NetLog and disk log, as implemented in the Thor system, was compared. Their performance was evaluated by running a benchmark that caused various amounts of data to be flushed to stable storage thus allowing a study of the variation in flush times with data size for the two log schemes. Results show that that the NetLog implementation that can survive 1 server failure performs better or as well as the disk log regardless of the flush data size. Thus, the increased availability provided by the NetLog comes for free without any decrease in performance. The NetLog may even provide improved performance as results indicate that the NetLog performs an order of magnitude faster than the disk log for flush data sizes less than 10 KB.

I also studied the performance of the NetLog implementation that survives two server failures. This implementation is faster than the disk log for small data sizes but when the data size is greater than 100 KB, the disk log becomes faster. However, this implementation of the NetLog is not efficient as it did not use the facility of multicast present in most local area networks. The NetLog performance in a network that supports multicast is studied in the model described in the next subsection.

1.3.4 Modeling the NetLog

The implementation of the NetLog and disk log abstractions in Thor provides one data point for comparison. To extend the comparison to more general system configurations, an analytical model of the NetLog was developed. The model allows the projection of the performance implications of the NetLog approach in future systems. The model was based on three different technological settings: CURRENT, ubiquitous technology today; FRONTIER, commercially available technology today; and FUTURE, technology a few years away. These three system configurations enable a comparison of the two log schemes with changes in the underlying disk, network and computer technology.

The results from the model are consistent with the results of the Thor implementation, namely that a NetLog that survives one failure performs better or as well as the disk log. From the model, it is seen that the relative performance improvement of the NetLog over the disk log is expected to get better as technology advances.

The model assumed a multicast medium with ordered acknowedgements when studying

the effect of varying the number of backups on flush time. Results from the model indicate that under such a setup, adding an extra backup introduces a small overhead (approximately 10% for a 10 KB flush). The model suggests that an optimal implementation of the NetLog that survives upto 10 failures can still perform better than the disk log.

1.4 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 details the NetLog abstraction, its architecture, interface and and operation. Chapter 3 describes the implementation of the NetLog within the Thor system along with its organization, internal structure and processes. Chapter 4 describes the setup, configuration, and experiments used to study the performance of the NetLog in Thor. Chapter 4 also presents the results of the performance comparison between the NetLog and the disk log in the Thor system. Chapter 5 extends the results of the performance comparison in Thor to future system configurations by developing an analytical model of the NetLog. The flush time of the NetLog and disk log are modeled to compare their performance. Chapter 6 makes some concluding observations and presents some areas for future work.

Chapter 2

The NetLog

This chapter describes the functionality and interface of a generic log abstraction and also details the design, organization and operation of network and disk based implementations of this abstraction. The chapter concludes with a look at the related work.

2.1 Log Abstraction

This section describes a generic log abstraction that can be used by any server application that requires access to stable storage. The log appears to the server as an abstract persistent data structure that permits the operations of addition, deletion and retrieval. The log abstraction has been designed to be independent of the application that uses it and of the nature of data stored within it.

Conceptually, the log is organized as a series of *log records*. The data that an application wants to store in the log is encapsulated within a log record. Every log record is associated with a unique *log index* represented by an integer between 0 and $MaxIndex$ (a large positive integer). A log record is allocated a log index when it is added to the log and the index of a log record does not change. Log indices are not reused and are visible to the applications for they are the means of distinguishing and accessing particular records within the log. The log has two pointers, *low* and *high*, that hold the indices of the oldest non-deleted record and the latest appended record in the log.

Physically, the log is composed of a volatile state and a persistent state. Each of these is organized as a series of log records. The data contained in the volatile part of the log is lost on system failure. The persistent part of the log will survive some system failure modes and be accessible when normal operation is restored.

The log abstraction provides applications with the functionality of adding, removing and retrieving log records as well as the ability to synchronize the volatile and persistent states of the log.

2.1.1 Adding Data to the Log

Log records are appended to the log using the *append* operation. When a log record is appended to the log, the log assigns the record its index and this log index is returned to the application that requested the append operation. When appended into an empty log, the log record receives the index 0. If the log is not empty, the log record is appended at the index which is one greater than *high*. After the log record has been added to the log, the append operation increments *high* by 1. The log record is appended to the volatile state of the log and thus appended records are not guaranteed to survive system failure.

Since append adds records at the *high* end and *high* does not wrap around, the log will run out of indices when *high* reaches $MaxIndex$. If *high* equals $MaxIndex$, the log will not permit any more append operations. The append operation might also fail because of lack of space. Implementations might have limited storage for log records and the append operation will fail if there is no space in the log to add the log record. Note that running out of indices and running out of space are independent of each other.

2.1.2 Removing Data from the Log

Log records may be removed from the log using the *delete* operation. Unlike traditional log abstractions that allow log records to be deleted only at the *low* end and in order, this log abstraction allows deletion to occur from anywhere within the log. This design is motivated by examining the requirements of many systems.

For example in the transaction manager coordinating the two-phase commit protocol in distributed databases [A94], the order in which transactions might commit may be different from the order in which their records were appended into the log. In a traditional log abstraction, the transaction that committed first cannot delete its log records till the other transaction commits and deletes its records from the log. Another use of this functionality is in the writing of updates in the log to the database [G95]. To maximize disk throughput, all the modifications on a database page should be written out together. These updates may be present in many non-contiguous log records and after the page has been written out to disk, all the corresponding records can be deleted.

The log consists of records with indices between *low* and *high*. In the traditional log abstraction that only allowed deletion from the *low* end and in order, the log guaranteed that all the indices between *low* and *high* contained valid log records. Allowing the functionallity of deleting any record implies that there can now exist indices, between *low* and *high*, which do not have a log record associated with them.

Applications delete log records by providing the index of the record to the deleted. If the record deleted was the earliest record in the log, *low* is incremented to the next higher index with a non-deleted log record. Like the append operation, the delete operation is performed on the volatile state of the log. The log records removed using the delete operation are thus still part of the log's persistent state.

Mutators:

```
Logindex append(Logrecord lr) throws LogFullException
//effects: Appends the logrecord lr to the log and returns the log index of lr within the log.
// Throws LogFullException if the log is full, due to lack of free space or indices.

boolean flush(Logindex i) throws LogEmptyException
//effects: Makes all the volatile records with log indices less than or equal to i persistent.
// Returns true if successful, else returns false. Throws LogEmptyException if the log is empty

void delete(Logindex i) throws NotFoundException, InvalidIndexException
//effects: Deletes the log record with index i from the log. Throws InvalidIndexException if i
// is out of bounds. Throws NotFoundException if there is no log record with index i.
```

Accessors:

```
Logrecord fetch(Logindex i) throws NotFoundException, InvalidIndexException
//effects: Returns the log record with logindex i. Throws NotFoundException if there is no
// log record with index i. Throws InvalidIndexException if i is out of bounds

Enumeration[Logindex,LogRecord] elements()
//effects: Returns an enumeration consisting of the logindex and log record of all the
// log records in the log in the order in which they were appended to the log, earliest
// to last.

Logindex high() throws LogEmptyException
//effects: Returns the log index of the last log record appended into the log. Throws
// LogEmptyException if the log is empty.

Logindex low() throws LogEmptyException
//effects: Returns the log index of the log record that was appended earliest among all
// the records in the log. Throws LogEmptyException if the log is empty.

int size()
//effects: Returns the number of records in the log.
```

Log Setup and Shutdown:

```
Configuration setup()

boolean shutdown()
```

Table 2.1: The interface to the log abstraction. The specification is written in Java-like syntax for illustration.

2.1.3 Synchronizing the Volatile and Persistent States

Since both the append and the delete operations act on the log's volatile state and do not survive system failure, the log's *flush* operation is used to synchronize the volatile and persistent states of the log. Specifically, the persistent state is updated to reflect the appends and deletes that have been performed on the volatile state. The flush operation takes a log index as an argument and all log records with indices lower than or equal to the log index specified are made persistent. The flush operation with argument i makes the effects of all operations that executed before it on records with indices lower than or equal to i persistent. Applications use the flush method to convey the effects of their modifications to stable storage.

By letting applications specify the record till which they want the persistent and volatile states to be synchronized, the log postpones flushing of records till it is required. Since flushing is a time consuming operation, this approach leads to significant advantages in a multi-threaded system, where a thread can flush the log only till the records it is concerned about. A thread that requests a flush of the records it appended to the log does not have to wait for the flushing of records that were appended by other threads after it appended its records but before it called the flush method.

The rationale for the separation of functionality between operating on the volatile state, and synchonizing the volatile and peristent states comes from studying application behavior and requirements. For example, the transaction manager discussed earlier appends a number of records into the log but requires persistence only after appending the commit record. This approach allows applications to implement performance optimizations to amortize the cost of flushing (usually a time consuming operation) over multiple append or delete operations [DKO84].

2.1.4 Retrieving Data from the Log

Since the log is independent of the application and the data stored, the log cannot interpret the data encapsulated within the log record. It therefore falls to the application to extract its log records and interpret the data stored within them. There are two ways in which applications can retrieve data from the log.

Applications can read data from the log by keeping track of their log activity and remembering the records that they appended and deleted along with their indices. The application uses the log's *fetch* operation that allows applications to read log records by supplying the index of the log record to be retrieved. The fetch operation acts on the log's volatile state and therefore will see the effects of all append and delete operations that preceded it. If there is no record at that index or the index is out of bounds (i.e. not between *low* and *high*), the fetch operation will fail.

Alternatively, applications may use the log's *elements* operation to iterate through the log records in the order in which they were appended to the log. This operation is particularly useful for applications that crashed and want to use the data in the log for recovery. The elements operation returns the list of non-deleted log records with indices between low and high. Since there may be log indices between low and high that do not

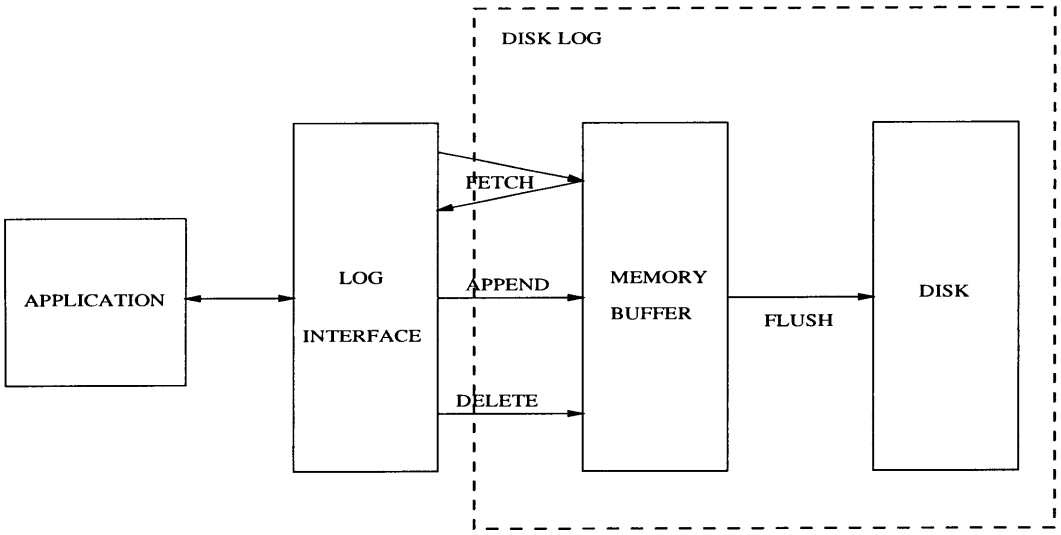

Figure 2-1: The Disk Log Architecture

have log records associated with them, the elements operation also returns the indices of the log records.

The log has a *size* operation that returns the number of records in the log. Note that since there may be indices between *low* and *high* that are not associated with log records, the number of records in the log is not the same as *high* - *low*.

2.1.5 Setup and Shutdown

The log's *setup* operation creates the log when the server is started and also recovers the log's volatile state from its persistent state. The setup operation returns a Configuration object that contains information about the configuration of the log like the amount of storage, value of $MaxIndex$, results of the recovery process.

The *shutdown* operation shuts the log down under normal case operation. It returns true if the shutdown was successful, otherwise it returns false.

2.2 Implementing the Log Abstraction

This section describes two implementations of the log abstraction described above: a disk log implementation and a network implementation. The interface to be implemented is described in table 2.1.

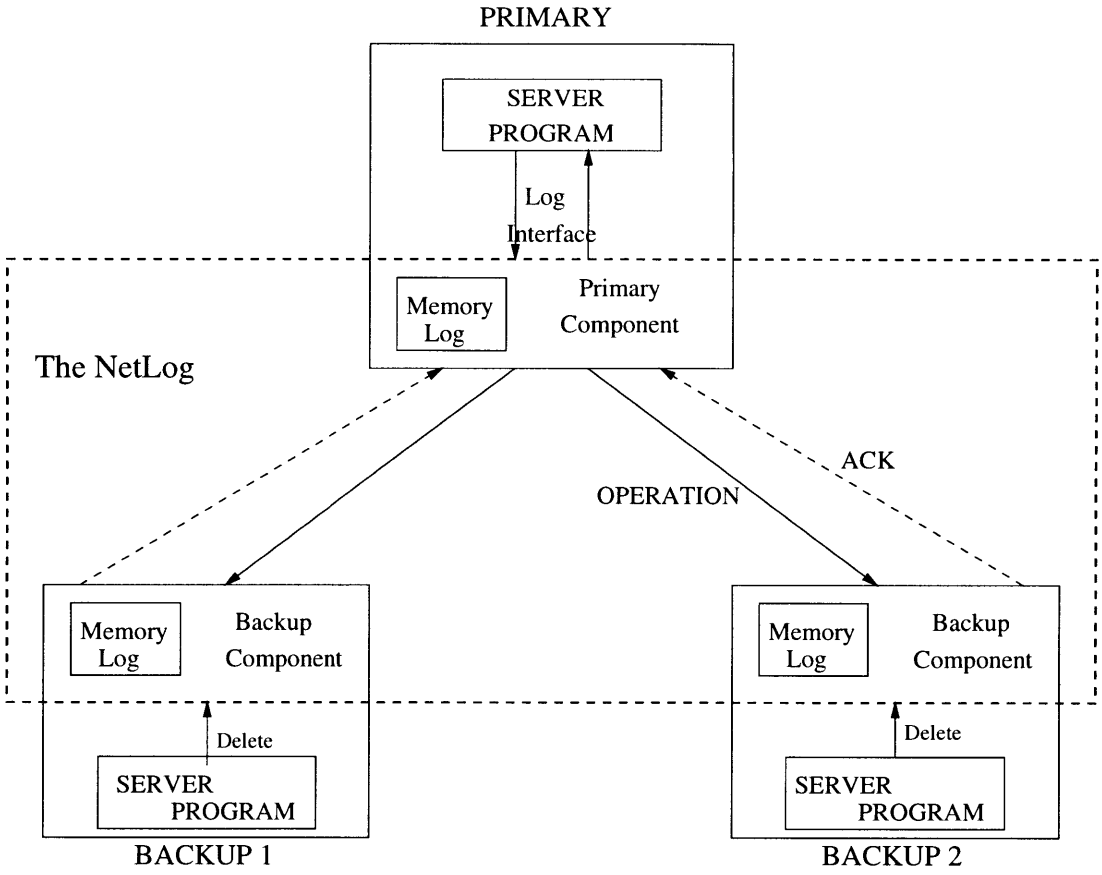

Figure 2-2: The NetLog System Architecture

2.2.1 Disk Log

The disk log implements the log abstraction using a re-recordable magnetic disk as the stable storage medium. The implementation uses both a memory buffer and a disk to store log records. The volatile state of the log is maintained in memory and the persistent state of the log is maintained on disk. The system organization is depicted in figure 2-1.

The append operation adds the log record to the memory buffer and the delete operation removes the record from the memory buffer. The flush operation writes the contents of the appended log records from the memory buffer to disk and deletes the records on disk that were deleted from the memory buffer. Fetch operations are satisfied in memory.

This implementation provides reliable storage for data by using the disk. However, if the server is down, the data is unavailable.

2.2.2 NetLog

The NetLog provides the same log abstraction in a distributed system thus allowing the data to be both reliable and available. The NetLog is a distributed program that runs on a number of machines in a distributed system, with a NetLog *component* running on each machine. Each machine is assumed to contain both memory and disk storage and

machines can communicate with each other through the network. This subsection details the organization and implementation of the NetLog.

Storage

Each NetLog component maintains a log data structure in the main memory of the machine that it runs on. Each machine is equipped with an Uninterruptible Power Supply (UPS) so that log data is not lost on power failure. If there is a power failure, the log data in memory can be written out to disk using the time afforded by the UPS. This data can be recovered into memory when the machine is booted up again and the NetLog component on that machine is restarted. Log data is stable because of the UPS and is available because it is present in the main memory of all machines. The principal advantage of the NetLog over the disk is that the log is available even when one or more servers are down.

Primary Copy Protocol

The NetLog uses the primary copy replication technique [AD76, O88, T79, G83] to manage the log records at all components. The machines are divided into a primary and backups. The server program at the primary interacts with the NetLog component at the primary through the interface described earlier (figure 2.1). The NetLog components at the primary and backups use the primary copy protocol to maintain synchrony and ensure that their log data is consistent. This is detailed in the operation section below. Figure 2-2 shows the architecture of a 2 backup NetLog configuration.

The organization of NetLog components (and the machines they run on) into a primary and backups is called a *view*. If a machine crashes, the NetLog components run a *view change* [ESC85, ET86, G83, O88] protocol. The result of the view change is a reorganization of the machines within the group such that the failed machine is removed from service. A similar view change occurs when a failed machine comes back up. In the new view created as a result of the view change, the machines that are part of that view are again composed of a primary and backups. The primary of the new view may be the same or different from the primary of the previous view. When the primary changes, the server program on the new primary interacts with the NetLog component at that machine through the interface in table 2.1. In a system with a primary and $2n$ backups, since the primary waits for responses from n backups, all persistent data is transmitted from one view to another, surviving upto n server failures.

With a group of $2n + 1$ machines, the NetLog can withstand upto n failures. Another interesting feature of the NetLog design is that in a set of $2n + 1$ machines, only $n + 1$ machines (the primary and n backups) need to have copies of the in-memory log data structure. The other n machines are called *witnesses*, as proposed by [P86]. The witnesses are required for the failover protocol to ensure that only one new view is selected. A witness also may play a part in the processing after a view change if it replaces a backup that has failed. The witness behaves just like a backup in the new view. When the failed machine comes back up, the witness ships the log records to that machine so that the failed server can re-synchronize with the primary of the new view.

NetLog Operation

The NetLog implements the append operation by appending the log record to the memory log of the component at the primary. When the NetLog receives a flush operation, the component at the primary sends the appropriate log records to the backup components so that the records can be inserted into their log. The backups then send acknowledgments to the primary component, as per the primary copy protocol described in section 1.2.1. In a system with $2n + 1$ machines, the primary waits for n acknowledgements before the flush operation completes. This ensures that the log data will survive upto n failures.

Unlike the disk log, the flush operation does not convey the records deleted at the primary. Since the NetLog is a distributed abstraction providing high availability, deleting the records at the backup as and when they are deleted at the primary will result in the log data being insufficient for another server to take over when the primary fails. Thus defeating the purpose of high availability provided by the log.

The delete operation for each component's log is the responsibility of the server program at that machine. When the server program at the primary issues a delete operation, the log record is deleted only from the log at the primary. This approach allows the backups to decide their extent of synchrony with the primary. If the backup wants to be closely synchronized with the primary, it will process the log records as soon as they arrive and delete them from the log. If the backup wants to keep the history of primary processing without being synchronized with the primary, it will accumulate log records and start processing only when its log is close to full or during a view change. However, fetch operations are carried out at the primary and deleted records are thus not visible to applications.

2.3 Related Work

One of the first implementations of the primary copy technique was the Tandem Non-Stop system [B78, B81]. The Auragen system [BBG83] was one of the first to combine the primary-backup scheme with automatic logging. The Tandem system had a *hot* backup where all the operations that the primary executes were executed at the backup. The Auragen architecture proposed a *cold* backup that, on primary failure, would use the log to come recover the system state and then take over. The NetLog takes an intermediate position where the log records reach the backup but may not be processed synchronously with the primary. The NetLog allows the server code at the backup to decide the extent of synchrony with the primary.

The closest work to the NetLog is that of Daniels et al [DST87]. They describe a system where there are a number of file servers that support client operations. The file servers share a distributed logging mechanism composed of a group of log servers that form a replica group for the log. The file servers use a quorum consensus scheme to write to the log servers. The file servers themselves are not replicated and if one file server goes down, another file server can query the log server to extract the log data in the failed file server and take over from the failed server. The crucial difference between the NetLog and their architecture is that their log is implemented on separate machines that must be accessed by

the servers using network communication. Further, their servers interact with the log using a quorum scheme which is more network message intensive than the primary copy scheme used in the NetLog.

The Harp file system [LGG91] contained a specialized, kernel-level implementation of some of the ideas in the NetLog design. Harp used the primary copy replication technique, with a view change algorithm, to achieve high availability.

The high latency of disk write operations to establish stable storage is avoided in the NetLog by storing the log data in memory and using a UPS. The idea of using a UPS to avoid synchronous disk writes also appears in [DST87, LGG91]. The Rio file cache [LC97] also uses a UPS to avoid disk writes in providing persistent storage to applications. The Clio logging service [FC87] achieves better write performance than re-recordable magnetic disks by using write-once media like optical disks for append-only log files. Another approach [BAD92] is to use Non-Volatile (NVRAM) based logs to prevent synchronous disk operations without losing information on power failure.

A software approach to ameliorate the disk latency problem by improving disk throughput is *group commit* [DKO84], which improves the throughput of disk operations by grouping together log records from a number of applications and writing them out in one shot. This approach utilizes the fact that most of the disk overhead is in setting up a write and once the disk is correctly positioned, disk throughput can be fairly high. The disadvantage of this scheme is that flush operations are delayed until a sufficient number of log records have accumulated.

The HA-NFS system [BEM90] also uses the primary copy scheme to provide high availability but since the log is kept on disk it suffers from the problem of expensive disk writes. The Cedar [H87] and DEcorum [KLA90] file systems also use disk based logs. Echo [HBJ90, SBH93] uses a primary copy technique with a log implemented on disk. The log disk is multiported and therefore is accessible by both the primary and backup and is managed by only one server at a time with the designated backup taking over when the primary fails.

Chapter 3

Implementation of the NetLog abstraction in Thor

This chapter details the implementation of the NetLog abstraction in the Thor distributed object-oriented database. The chapter begins with an overview of the Thor system, highlighting the role of the log in the Thor server. The chapter then describes the NetLog structure and operation in the Thor system. The chapter concludes with a discussion of the disk log implementation.

3.1 Overview of Thor

Thor [LAC96] provides a universe of persistent objects that are created and used by applications. Applications access Thor objects as they would normal, non-persistent objects provided by the programming language in which they are written. Thor is based on the client-server paradigm with applications running on clients and the persistent object store being distributed among the servers.

Applications access Thor objects by starting a *session.* Within a session, applications perform a sequence of *transactions.* Each transaction is composed of a set of more elementary operations, like reads and writes. The system guarantees that the entire transaction succeeds (commits) or the entire transaction fails (aborts). If a transaction fails, the system guarantees that none of the operations executed within that transaction has any effect on the database. If the transaction commits, all the operations performed by that transaction are guaranteed to survive all subsequent failures upto some limit.

Applications run transactions at the client machines. The client maintains a cache [D95, CAL97] of recently used objects and fetches objects as needed from the server. It also tracks the objects read and modified by the current transaction and sends this information to the server as part of the commit request when the transaction completes.

When a server receives a commit request, the request includes the new objects to be made persistent and the new values of all objects modified by the transaction. The transaction and the modified objects are first vetted through the concurrency control scheme [A94], which

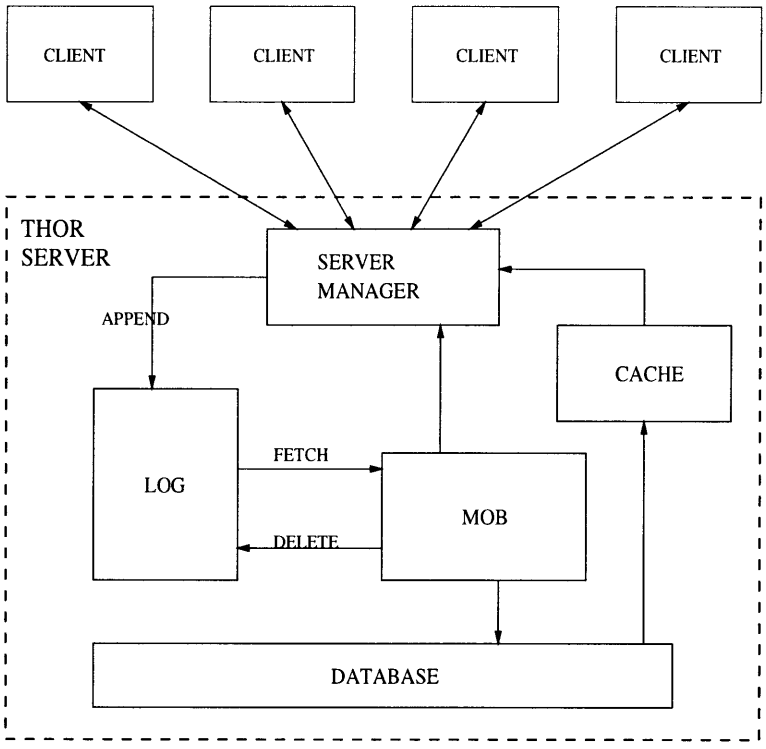

Figure 3-1: The Thor Server and the Log

ensures that the transaction does not violate the sharing semantics defined by the system. If the transaction does not cause any sharing violation, the transaction is allowed to commit; otherwise it is aborted. If the transaction is allowed to commit, the system then writes the new values of the modified objects to the log. The transaction commits successfully if and only if all of the modified objects reach the log. If the write of these objects to the log fails for some reason (server crash, stable storage error, etc.), the transaction is aborted. The server notifies the client about the commit as soon as the write to the log is completed.

Each Thor server uses the disk to provide persistent storage for the database. When the server receives a fetch request from a client, it reads the pages of the requested objects from the disk. The server maintains a cache of objects to avoid going to the database on disk repeatedly for the same objects. The server also maintains another in-memory data structure called the modified object buffer (MOB) [G95], that is used as a repository for objects modified by recent transactions; the MOB is used to satisfy read requests for these objects. The MOB management program writes the new and modified objects to the database and deletes the corresponding records from the log and the MOB.

The role of the log in the Thor server along with the Thor architecture is shown in figure 3-1. Apart from its use in transaction management, the log is also used by other system processes, like the garbage collector [M97], to ensure that the system operates consistently

in the face of server failures.

3.2 NetLog Implementation

This section describes the implementation of a NetLog consisting of a primary and one backup. The primary and backup each run a program called the *log* that performs the actions of the NetLog component (section 2.2.2) on that machine. The primary and backup log programs run different code but maintain similar data structures.

3.2.1 Internal Representation

The log program is organized as a data abstraction and all operations are performed by calling the methods of this abstraction, thereby ensuring that the internal representation of the log is not exposed.

Data Structure

The log program, either at the primary or the backup, maintains log information in a circular list of log entries, `list`, in the main memory of the server on which it runs. The number of entries in the list, `size`, can be changed dynamically. The variable `start` specifies the earliest entry in `list` and `count` specifies the number of valid entries in the list.

Log Entries

Each log entry in the `list` is just a wrapper for the log record that it contains. Each log entry contains a pointer to a log record present in the heap and a flag specifying if the record has been deleted. A log record is a composite data structure that contains a number of fields and may be one of several types depending on the purpose for which the record has been appended to the log. The number, type and interpretation of the fields that a log record contains is dependent on the type of log record. Note that some log records may contain fields that are pointers to other data objects in the heap. An example of this is the Commit record, which is appended to the log when a transaction commits, and contains pointers to all the new/modified objects of that transaction. Figure 3-2 shows the internal structure of the log. Since the semantics and structure of the log records are different, the log is oblivious to the details of the log records and interacts with the log records through a specific interface.

To implement that interface, all the different types of log records are subclasses of an abstract class, `LogRecord`, which contains the following abstract methods that are implemented by each subclass. A `size` method is present that returns the size of the log record which is simply the sum of the sizes of its fields. Note that this does not include any objects that are pointed to by fields within the log record. Each log record has a `encode` method that returns all the data represented by the log record as a bit stream. This includes all the objects that are pointed to by fields within the log record. For example in figure 3-2, the

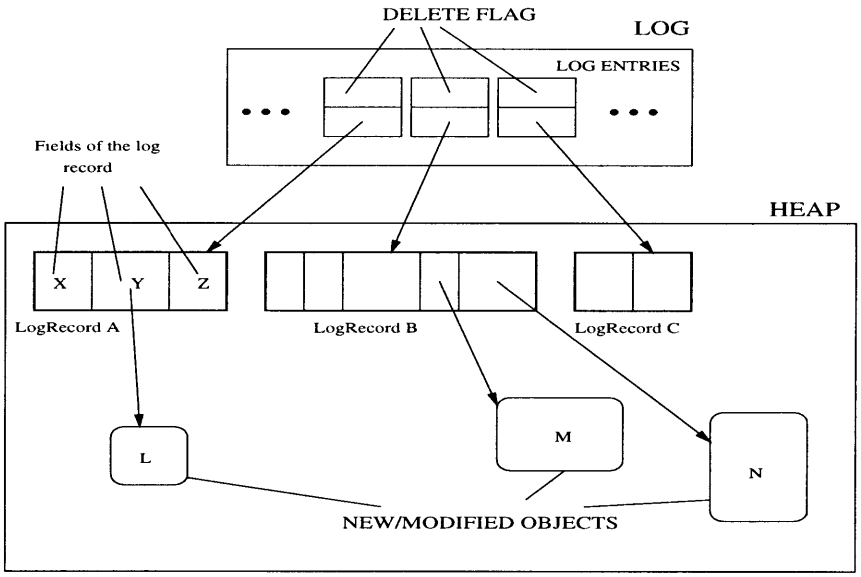

Figure 3-2: Log Records and the Heap

bit stream would contain, in order, the encoded representations of field X, field Y, object L and field Z. Complementary to the `encode` method is a `decode` method that when given a bit stream will reconstruct the log record with appropriate fields and pointers thereof. The decode method uses the heap to create external data objects pointed to by fields of the log record.

Log Indices

As described in section 2.1, there are two types of records in the primary's log: log records that are stable, and those that are not. The stable log records have been flushed to the backup's log. The primary's log keeps track of which records are stable using the following three variables. The log variable, `next_index`, specifies the index of the next to-be-inserted log record; `next_flush` denotes the index of the last log record flushed to the backup; and the log variable `low` denotes the index of the starting element of the primary's log.

At the backup's log, the variables `low` and `next_index` have similar meaning. The backup does not maintain a `next_flush` variable.

Note that `next_flush`, `next_index` and `low` hold the indices of log records that are visible to applications. The log, at both the primary and backup, performs a mapping from its internal storage of the records in the `list` structure to log indices. The index of the record pointed to by the log entry in `list[start]` is `low`. The log entry `list[(l-low+start) mod size]` contains a pointer to the log record with index `l`. The log maintains the following invariants: `size` $\geq$ `count` ≥ 0 and `next_index` $\geq$ `next_flush` $\geq$ `low`.

Since the log record indices do not wrap-around, a log record is identified by the same

index at both the primary and the backup. However, n^{th} log entry in the `list` data structure need not point to same log record at both the primary and the backup.

3.2.2 Code Structure

The primary log is used by the following processes. The log is created by the server process that initializes all server components. The transaction manager on the primary forks a thread (one for each client) that performs the operations of the transactions from that client. These transaction threads append log records to the primary's log. Log records are read and deleted by a MOB thread that writes the modifications described in the log records to the database. The log is shutdown by the process that shuts down the Thor server on power failure or for routine maintenance.

The network handler for messages from the primary appends records to the backup's log. The log records are read and deleted by the MOB thread that maintains the database at the backup.

At either the primary or the backup, there are a number of threads that access the log concurrently and in order to ensure that all log operations are ordered and do not interfere with each other, access to the log is controlled by an external `mutex`. A mutex is a special variable that can be acquired by only one thread at a time. The thread that holds the mutex can perform its operations on the log and once these operations have been completed, the thread relinquishes control of the mutex so that it can be acquired by the next thread that wants to access the log.

Flush Thread

The primary's log has a thread running in the background called the `flush thread`. The `flush thread` controls access to the network connection to the backup and is responsible for sending data to the backup. The log's flush method (section 3.2.3) signals the flush thread to send the data to the backup and waits for a signal from the flush thread that it has finished before returning. There are two main reasons that a separate flush thread was implemented.

The first reason is to permit spontaneous flushing of log records so that the server does not have to wait for the flush to complete. For example, in a system that performed incremental logging of the data, by the time the flush operation is called, the log may have accumulated a lot of data and flushing all of the data may be time consuming. A flush thread could proactively flush data so that when the flush call does come in, it can return much sooner since most of the flushing may have already been done.

The second reason is to permit batching of log records, from a number of processes that are appending records to the log, into one flush. This is useful under conditions of heavy load when there are a number of transaction threads, all of which want the log to be flushed. Under heavy load, many transaction threads will get to run before the flush thread and all of them will be signalling the flush thread. The flush thread is thus woken up once and can perform the flushing for a number of threads in one shot. Since flushing involves

a significant setup cost, this approach allows the setup cost to be amortized over a larger number of records thereby increasing throughput.

Backup Flush Thread

The backup's log has a thread called the `backup flush thread` that runs in the background and is responsible for getting the log records from the primary and appending them to the log at the backup. Since the log records arrive through messages from the primary in an asynchronous fashion, having a separate thread separates the processing from the main server threads at the backup that are not involved with the appending of log records.

3.2.3 NetLog Operation

This subsection describes how the NetLog (consisting of log programs at the primary and backup) provides the server at the primary with the functionality described in table 2.1.

Setup

Though the backup might be used for other purposes, the current implementation of the NetLog assumes a dedicated backup server. The primary log program is started by the primary process that creates all server components.

The primary log program initializes the `list` data structure along with all log variables. If the log is being re-started from a previous configuration, the state of the log is recovered from a special region on disk (see shutdown section below). The primary performs an RPC to create the log program on the backup server. The configuration information, i.e. the servers on which the primary and backup logs are placed, the addresses of these servers, and the port numbers that the primary and backup use for establishing network connections, can be changed in the file `replication.setup` and does not require recompilation.

The backup log program first initializes the log data structures and recovers its state from disk, if it is being restarted from a previous configuration (see shutdown section below). After initialization, it forks the `backup flush thread` which binds and listens in on a pre-specified port number, waiting for the primary to establish a connection.

After the RPC calls have returned, the primary waits for 5 seconds before forking the `flush thread`. These five seconds are for the log at the backup to be properly set up and the `backup flush thread` to be ready to accept connections. The `flush thread` on the primary then synchronizes with the backup and establishes the network connection.

After the two flush threads have established a network connection, they set up the message handlers. The `backup flush thread` sets up message handlers for receiving flush data messages and shutdown messages from the primary. The `flush thread` at the primary sets up message handlers for the acknowledgement messages from the backup.

Append

The NetLog append method appends records to the log at the primary. The log record to be added to the log is heap allocated by the appending process and the pointer to the log record is handed to the append method.

The append method first gains control of the primary log mutex. It then checks if the `list` structure is full or not. If so, the `list` is enlarged to contain more log entries and the pointer to the log record is inserted within the log entry at `list[(start+count) mod size]`. The record receives the log index `next_index` and both `next_index` and `count` are incremented by 1. After the mutex is released, the index of the log record is returned to the appending process.

Flush

The NetLog's flush method sends the data in the primary log to the backup logs. As described in table 2.1, transactions threads will call the flush method with a log index as an argument and expect all records with indices lower than that to be flushed to the backups when the flush method returns.

The flush method first checks if the index specified is less than or equal to `next_flush`. If it is, the record has already been flushed and the flush method returns immediately. If not the flush method signals the flush thread at the primary (which has been sleeping waiting for something to flush) and then blocks on a condition variable waiting to be woken up by the flush thread when the flush has been completed. If there are a number of transaction threads that signal the flush thread simultaneously, they all block on the same condition variable.

The flush thread first acquires the log mutex and identifies all the records in the primary log that need to be flushed by a linear scan from `next_flush` to `next_index`. Once this is done, the value of `next_index` is stored and the flush thread relinquishes control of the mutex. The value of `next_index` needs to be stored so that `next_flush` can be updated once the flush is completed. This is required as `next_index` might have been changed by other transaction threads appending records into the log before the flush operation completes.

Once the log records have been collected, the flush thread calls their encode functions and aggregates the bit streams into one message which is sent to the backup. When a log record arrives at the backup, it has to be stored into some data structure. Since the `LogRecord` class is an abstract one, the backup needs to know the type of the log record that is coming on the wire in advance so that it can create the appropriate storage for it. The primary therefore inserts the type of a log record before its bit representation in the flush message. At the backup, the processing for each message is done at the level of the message handler itself. The message handler, on reading the type of log record, heap-allocates storage for it and uses the log record's decode method to reconstruct the log record in the heap. The log record is then appended to the log using the append method.

Once the above operations have been completed, the message handler at the backup sends an acknowledgement to the primary with the index of the last log record received.

When the acknowledgement from the backup reaches the primary, the flush thread at the primary obtains control of the log mutex so that the `next_flush` value can be updated with the value of `next_index` stored before the start of the flush. The flush thread then wakes up the transaction threads that signalled it and are waiting for the flush to be completed. The flush thread then goes back to sleep.

The transaction thread, when woken up by the flush thread, continues execution of the flush method. It then checks whether the log index till which it requested the flush is lower than or equal to `next_flush`. If it is so, the flush method returns. Otherwise the flush method will signal the flush thread and the entire process will be repeated.

Delete

The NetLog delete operation takes the index of the record as an argument and deletes the record from the primary's log. The delete operation obtains control of the log mutex and marks the delete flag in the log entry as deleted. If the record that was deleted had index `low`, the value of `low` is incremented to the next greater index with a record associated with it. The variable `start` is also similarly updated and the value of `count` is decremented by one. After releasing the mutex, the delete call returns.

When a transaction commits, the effects of the transaction, as described in the records it appended to the log, must be applied to the database. There is a MOB thread that runs in the background which writes the modifications described in committed log records to disk and frees up the storage connected with these log records. The MOB thread then calls the log's delete method to let the log reclaim the associated log entry.

Fetch

The NetLog fetch operation obtains control of the mutex and, if the index is within bounds (between `low` and `next_index`) and the record associated with it is not deleted, the fetch method returns the log record. After releasing the mutex, the fetch method returns.

High

The NetLog high operation returns the index of the last appended record. The high method obtains control of the mutex at the primary, stores the value of `next_index` in a temporary variable and after releasing the mutex, returns the value of the temporary variable minus 1.

Low

The NetLog low operation returns the index of the earliest non-deleted record. The low method obtains control of the mutex at the primary, stores the value of `low` in a temporary variable and after releasing the mutex, returns the value in the temporary variable.

Shutdown

If the log is shutdown at a server, the log's state consisting of state variables, `list` structure, log records and all associated heap objects, is written to a special region of the disk. If the system wants the log to be restored to a previous configuration, the log's setup method checks for data in this region and recovers the state of the log at this server.

If the log is shutdown as part of server maintenance or some other user-controlled activity, the process shutting down the server calls the log's shutdown method. The primary log sends a shutdown message to the backup. When this message arrives at the backup, the shutdown message handler activates a thread on the backup log that writes all log data to the special region on disk. The shutdown message handler then sends an acknowledgement message to the primary. The backup then kills all threads and shuts down the backup server. When the primary receives the shutdown acknowledgement from the backup, the primary writes all the data in its log to the special region on disk. The primary is then also shut down. Note that the primary doesn't have to wait till the backup has shutdown before writing its records to disk but this is done to ensure that in case of any failure in shutting the backup down, the data required to restore the backup is still in memory so it can be accessed easily.

If the log is shutdown due to power failure or some other abnormal condition, the log's shutdown method is called by the server process handling this event. The log just writes all the data in the log to the special region of disk, kills all threads and returns.

3.3 Extending the Base Implementation

The previous section focussed on a NetLog implementation with one backup to which log data is flushed. Using a witness that is only used for failure processing, this implementation survives one server failure. NetLog implementations that survive more failures are simple extensions of the base case. To demonstrate this, an implementation that survives two server failures was developed. Another backup log on a different server was added. The main change in primary behavior is that the primary sends out flush messages to the two backups one after another and waits for acknowledgements from both of them before proceeding. In this implementation, multicast was not used to send log records to the backups and the effect of a multicast medium on performance is discussed in section 5.8.

3.4 Disk Log Implementation

A disk log was also implemented in Thor to compare the performance of the two log abstractions (chapter 4). To subsitute one log with the other seamlessly, a device abstraction was created. The log methods interact with stable storage through the methods of the `Device` abstraction, which could either be a network connection to a backup or a data structure on disk.

The `Device` Abstraction

In order to allow one abstraction to be substituted for the other, both the network and disk devices were made to inherit from the abstract class `Device`. The functions that the log methods use are abstract functions of `Device` class. If it is a NetLog implementation, the setup operation will create the network connection to the backup and encapsulate the network connection within a class that inherits from the abstract class `Device` and that implements the abstract methods of the `Device` class. Similarly, in the disk log implementation, the disk is encapsulated within a class that inherits from `Device` and implements the abstract methods of `Device` that are used by the log methods. This arrangement allows the log to be oblivious of the medium to which the flush is directed.

For the NetLog, a log record has reached stable storage if and when it has been inserted into the log at the backup. The write to the `Device` abstraction returns when the server receives an acknowledgement from the backup. For the disk log, the log record has reached stable storage when the record has been written to disk successfully and the write to the `Device` abstraction returns when the disk write completes.

Disk Log Operation

The disk log implementation is similar to the NetLog implementation, the main difference being in the medium to which the flush operation is directed. The disk log is implemented as shown in figure 2-1.

As in the NetLog implementation, the log is created through the setup operation when server is started. When the log is created, the log data structure is created and initialized on the disk. Similar to the NetLog implementation, if the server is being restored to a previous configuration, the setup method checks for data in the special region of disk and restores the log state. This disk structure is cast into the `Device` abstraction as described above.

The append, high, low and delete operations are identical to the NetLog, since they are performed on the log data structures in memory. The flush operation synchronizes the state of the primary in-memory log data structure with the log on disk. When the disk log is shutdown, all the log data in memory is written out to the special region on disk so that the log can be restored to the same state when it is restarted.

A problem with the current implementation of the disk log is that there is no dedicated log disk on the systems that Thor runs on. A dedicated log disk was simulated by careful placement of files and ensuring that only log activity takes place on one disk. This is elaborated in section 4.1.4.

Chapter 4

Performance of the NetLog in Thor

The NetLog provides servers with a highly available, stable storage abstraction. Magnetic disks, used by many systems to implement reliable stable storage, do not provide high availability for the data stored in them, without significant hardware and software extensions. This chapter examines whether the increased availability provided by the NetLog has any effect on performance by comparing the performance of the NetLog and disk log implementations in the Thor system. In order that the results be extendible, the performance of the log is characterized on a metric that is independent of the system in which it is implemented. The performance metric that is used for comparison is flush latency, i.e., the time required to flush log records to stable storage. The flush time is studied as a function of the size of the data to be flushed.

4.1 Experimental Setup

This section details the experimental setup used for measuring the flush times. The experiments were run using a client program and a server program, with the client performing transactions on the server which managed the database. The server program used the log as described in 3.1 and flush times were measured at the server.

4.1.1 The OO7 Database and Traversals

The experiments presented this chapter are from the flushing of log records appended to the log by OO7 transactions on the OO7 database as described in [CDN93]. The OO7 benchmark has been developed to model complex CAD/CAM/CASE applications built on top of an object-oriented database. The results described in this chapter do not depend on the OO7 benchmark for the flush latency was studied as a function of only the flush data size, independent of the database, transaction, or flush data. The OO7 benchmark was used because it provided a ready testbed for measuring flush times.

The OO7 database consists of a tree of *assembly objects* with the leaves of the tree pointing to three *composite objects*. Each composite object is actually a graph of *atomic objects* that are linked by *connection objects*. There are two kinds of OO7 databases that I run experiments on: the small and the medium. The small OO7 database has 20 atomic objects per composite object, while the medium OO7 database has 200. The size of the small database is 4.49 MB and that of the medium database is 39.72 MB.

The OO7 benchmark describes a number of traversals (transactions) that can be run on the database. Traversals vary in their effects on the database and the size of the log records that they cause to be appended into the log. For the purpose of this thesis, two traversals are of interest:

- *Traversal 2a*: This traversal modifies only the root atomic part of the graph and it requires a flush to stable storage of 2 records that contain the modifications. The total flush data size is about 23KB for both small and medium databases.
- *Traversal 2b*: This traversal modifies all the atomic parts of the graph and it causes two records to be appended into the log. The total size of the log records is about 471 KB for the small database and about 4.7 MB for the medium database.

Since the purpose of this chapter is to compare the performance of the two log implementations, the nature of the transaction that created the log record is immaterial. What is important is the size of the record appended and the variation of the flush times for both schemes with log record size. Hence, a modified transaction that is an intermediate between T2a and T2b was created that permitted varying the size of the log records created by changing a parameter in the transaction. This parameter controlled the percentage of atomic parts within each composite part that was changed by the hybrid transaction. When this parameter is zero, the hybrid transaction behaves like the T2a and when this parameter is 100%, the hybrid transaction behaves like the T2b. By using this hybrid transaction, the variation of flush time with flush data size was studied.

4.1.2 System Configuration

The experiments were run on DEC 3000/400 workstations. Three configurations were used.

- *Disk*: This configuration was used to measure the flush times for the disk log. Client and server programs were run on the same workstation.
- *Network 1B*: This configuration was used to measure the flush times for a NetLog implementation that can survive one server failure. The client and primary server programs were run on the same workstation and the backup server program was run on another identical workstation.
- *Network 2B*: This configuration was used to measure the flush times for a NetLog implementation that can survive two server failures. The client and primary server programs were run on the same workstation and the two backup server programs were run on two other identical workstations.

Due to limitations in system configuration and resources, the client and server had to be run on the same workstation. This obviously had a detrimental effect on system performance when compared to other published results of the Thor system [LAC96] or the results of the model (chapter 5). Since the purpose of this thesis is to examine the *relative* performance of the two log abstractions, the above configurations ensure that the degradation in performance is seen by both log implementations.

4.1.3 System Characteristics

The workstations had 128 MB of memory and ran Digital Unix version 4.0. In the Network configurations, the workstations were connected by a 10 Mbps Ethernet and had DEC LANCE Ethernet interfaces. The disks on the workstations were DEC RZ26, with a peak transfer rate of 3.3 MB/s, average seek time of 10.5 ms and an average rotational latency of 5.5 ms. The Thor source code was compiled using the g++ compiler and all experiments were run in an isolated setting.

4.1.4 Simulating a Dedicated Log Disk

Since none of the workstations had a dedicated log disk, careful placement of files was used to simulate a dedicated log disk. The log was placed on one partition of a disk and no other activity (system or user) was allowed to take place on that disk. As in dedicated log disks, there is no interference with the logging process but there is a lot of data already present in that partition of the disk. Even though only log data is streamed to the disk, the data is not written out contiguously on the disk as detailed in section 5.4. The head is, therefore, occasionally repositioned from track-to-track due to the fragmentation of the disk. In the experiments, the disk log was placed in a partition with 100MB of free space thus mitigating but not eliminating this effect.

4.2 Results

Figure 4-1 shows the variation of the flush time for the disk and NetLog abstractions with flush data size. Logarithmic scales are used for both axes. It is seen that the disk log has a very high fixed cost and a much lower variable cost. This behavior is consistent with the fact that the every disk write involves a fixed setup cost, namely the positioning the disk head over the right position to write. After the head is positioned correctly, disk throughputs are very high leading to low variable cost. For the NetLog, the costs of sending messages are roughly proportional to the size of the message as network per-message-set-up times are small. Note that the slope of the NetLog is initially small and but increases after flush data size cross 1500 bytes as this is the packet size of the ethernet network connecting the servers. Flush sizes less than this are accomodated in one packet leading to lower variable cost.

The experiments varied flush data sizes from 80 bytes to 4.5 MB and the Network 1B system was always faster than the disk log implementatation. The disk and Network 1B

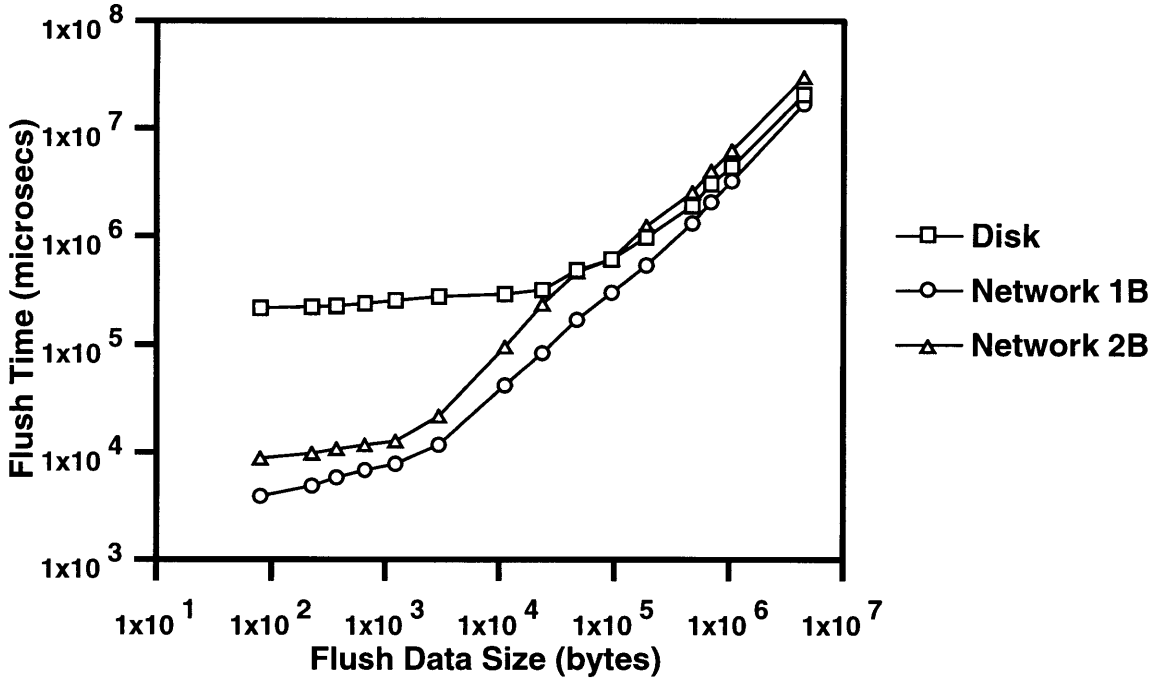

Figure 4-1: Variation of flush time with flush data size.

curves display almost identical behavior when the data size is increased further. Note that the NetLog is an order of magnitude faster for data sizes less than 10 KB and twice as fast for data sizes upto 100 KB. For a flush data size of about 4.5 MB, the disk was about 18% slower than the NetLog. The Network 1B system can survive one server failure and provides high availability compared to the disk log without any decrease in performance. In fact results show that it may lead to an improvement in performance, especially for small flush sizes where the speedup may be an order of magnitude.

When the disk log is compared with the Network 2B configuration, the NetLog is an order of magnitude faster than the disk log for data sizes less than 1 KB. As flush size increases, the disk gets progressively better to when it is faster than the Network 2B system for flush data sizes greater than 100 KB. The Network 2B system remains available even after 2 server failures while the disk is unavailable on server failure. For flush sizes lower than 100 KB, increased availability comes for 'free'. For flush sizes greater than 100 KB, increased availability comes at the cost of decreased flush times. After about 1 MB, the Network 2B system is about 30% slower than the disk log. Note that the Network 2B implementation was not ideal as it did not use multicast to communicate to backups. The effect on flush times of using multicast for primary-to-backup communication is described in 5.8.

4.3 Conclusion

This chapter detailed the performance of the disk log and NetLog implementations that could survive one and two server failures in the Thor system. A large range of flush data sizes were studied and it was seen that in the current Thor implementation, the NetLog that survives one server failure always performs as well or better than the disk log. The Thor results indicate that the NetLog not only provides higher availability than the disk log but that this availability does not result in a decrease in performance; and it may actually result in an improvement in performance for small data sizes. In the next chapter, a detailed model of the disk and network flush process is developed in order to extend the performance comparison to other system configurations.

Chapter 5

Modeling the NetLog

In this chapter I present an analytical model that estimates the latency of the flush operation in both the NetLog and the disk log implementations. Flushing data to the disk log involves writing a block of data to a dedicated log disk on the same machine. Flushing data to a NetLog involves sending the data to machines across a network and waiting for an acknowledgement from those machines, after they have inserted the data into their in-memory log.

In the previous chapter, I compared the flush times of both log abstractions as implemented in the Thor system and found that a NetLog that survived one failure had as good or better flush times than a disk log indicating that the increased availability provided by the NetLog did not affect performance. The purpose of developing the model is to compare the performance of the two log abstractions on different system configurations. The model thus allows an assessment of the relative performance under the impact of advances in disk and networking technology.

The model developed in this chapter is analytical as analytical models are easily developed, manipulated and evaluated. However, studying the behavior of disk and network systems requires detailed simulations and analytical techniques are not accurate. In this thesis the metric that I compare the two log abstractions on is average flush latency. The key point to note is that since only the simple *average* of the flush latencies is estimated, the distribution and behavior of the flush latencies are not important. This enables an analytical model to approximate the results from simulation while being much simpler to develop. The model developed in this chapter extensively uses results from numerous published simulation studies to ensure that the approximation is a good one. Contention for system resources, like disk, network and CPU is not considered for the model assumes dedicated system resources.

The model uses some system parameters in estimating the flush latency. While these values vary from system to system, values from the Thor system are used in calculations so that the results of the model can be compared to the performance results described in chapter 4.

Except in section 5.8, references in this chapter to NetLog refer to the NetLog configuration that survives one server failure.

5.1 CURRENT, FRONTIER and FUTURE Systems

At the heart of the model is an assumption about the nature of the underlying technology of the system in which these log abstractions are embedded. I examined three different settings: CURRENT, FRONTIER and FUTURE. These settings are for basic machine and system characteristics. The CURRENT system represents the state of technology that is widely deployed today. It also reflects the hardware that the current version of Thor runs on. The FRONTIER system represents technology that is commercially available today (1998). The FUTURE system represents my calculated estimate of technology to be widely available a few years into the future (2002). These three settings were chosen so that the relative performance of the log abstractions could be assessed over time and with changes in the underlying network and disk technology.

5.2 Platform settings

In computing the flush cost for each of the three system settings discussed in the previous section, I used data from a number of published studies of the disk and the network. However, since these studies were conducted on a variety of processor configurations, results from different studies cannot be coalesced without accounting for the differences in processing speed. I used their system's performance on the SpecInt92 [S92] benchmark as the means of scaling processor speeds from their system configuration to the system configurations described in the previous section. The reason for choosing the SpecInt92 over other benchmarks is that it is used by all the studies from which I draw data [G97, MB93, VBB95] and furthermore, data for the various hardware platforms of relevance to this study were readily available for this benchmark. The CURRENT setting was based on the system that the current version of Thor is implemented on, namely, the DEC Model 3000/400 133 MHz 21064. The FRONTIER system was based on the DEC Alphastation 600 5/266. The FUTURE system's SpecInt92 rating was set at 500.

5.3 The Model

I use a simple model of flushing cost, which is measured in units of time. The total cost of flushing data is assumed to be composed of a fixed cost and a variable cost. Every flush operation has a fixed cost associated with it regardless of the size of the data being flushed. I call this fixed cost the latency, L, of the flush operation. The speed of every flush operation is also limited by the rate at which data can be flushed. The more data there is to flush, the larger the flush cost. The byte flush cost, b, of the flush operation is the cost of flushing one byte of data. The variable cost associated with a flush of M bytes of data is, therefore, the product of the byte flush cost and the size of the data. The total flush cost, F, for M bytes of data is therefore

$$F = L + M * b$$

The flush costs for any system to which flush operations may be directed can be com-

	CURRENT	FRONTIER	FUTURE
Full Rotational time frt (ms)	11.1	5.8	3.0
Data Transfer Rate B (MB/s)	2.7	17.9	50.0
Track to Track switch t (ms)	2.5	1.0	0.5
Track capacity s (KB)	38	89	109

Table 5.1: Disk System parameters for all three settings.

puted using the above model by estimating the system's latency and byte flush cost. In the next two sections, I will model the disk log and the NetLog using this approach by estimating the latency and byte flush cost for each.

5.4 The Modeling the Disk Flush

Disk drives are composed of two main components: *mechanism* and *controller*. The mechanism is made up of the recording components (rotating disks and the heads that access information from them) and positioning components (the arm assembly that moves the heads onto the correct track, along with the track positioning system that keeps the heads properly positioned on the track). The disk controller is made up of a microprocessor, buffer memory and the interface to the SCSI bus. The controller is responsible for managing the data flow (read and write) from the host system to the disk and is also responsible for mapping logical disk addresses, which the host system sees, to physical disk addresses.

I model a dedicated log disk. This disk is used only for logging and is used exclusively to store log records. The flushing cost in a dedicated log disk is composed of four major components: CPU activity to setup the disk write, disk controller overhead in preparing to write data, delay in positioning the head, and data transfer time.

In developing this model, I drew heavily from Thekkath et al [TWL92] and Ruemmler and Wilkes [RW91, RW93, RW94]. These papers provide very detailed models of disk drives and include extensive simulation results that were useful in understanding the extent of the error involved in using an analytical model. The HP C2247 [HP92] was used as the reference for the CURRENT system. The Seagate Cheetah 18 [S97] drive is the high end of the Seagate product line and was used as the reference for the FRONTIER system. The FUTURE system was extrapolated from these two system settings. Table 5.1 describes the disk system settings.

5.4.1 Latency

The latency, L, associated with a disk flush is composed of CPU costs, disk controller overhead and rotational delay in head positioning.

CPU costs, C, are based on the number of instructions that the host system has to execute to set up the disk write. Gruber [G97] and Carey et al [CFZ94] provide estimates of the CPU cost for setting up a disk write and these costs were scaled for the CURRENT,

FRONTIER and FUTURE systems as described in 5.2.

The disk controller manages access to the SCSI bus and the SCSI bus interfaces at the host and the disk take time to establish connections and decipher commands. In a detailed study, Worthington et al [WGP94] provide values for many of these low level overheads on the HP C2247 drive and these were used for the CURRENT system. For the FRONTIER and FUTURE systems, Gruber and Rummeler and Wilkes report that disk drive technologies are advancing at approximately 12% compounded annually and I assumed this rate to compute the controller overheads for the FRONTIER and FUTURE configurations. [WGP94] was conducted in 1994 and values for the FRONTIER and FUTURE systems were calculated given that they are set in 1998 and 2002 respectively.

Head positioning typically consists of seek time (time to position the head over the correct track) and rotational latency (time to wait for the disk to spin till the head is over the correct position on the track). Seek time will be addressed as part of the byte flush cost in the next subsection. In their experiments, Ruemmler and Wilkes found that estimating the rotational time by a random selection between 0 and the full rotational time, frt, instead of their detailed simulations increased the demerit figure in their experiments from about 2% to about 6% of the actual disk performance. For the purposes of this thesis, this margin of error is acceptable and an average rotational delay of half the full rotational time was added to each disk write. The rotational delay for the C2247 and the Cheetah are described in their datasheets and the trend between them was extrapolated to the FUTURE system.

The latency associated with a disk flush is therefore

$$L = S + C + \frac{frt}{2}$$

where S is the disk controller overhead.

5.4.2 Byte Flush Cost

Data transfer costs measure the time taken for the actual writing of the data onto the disk once the write has been setup at the host and controller level and the head is positioned correctly. The data transfer cost depends on the amount of data being flushed and can be computed knowing the number of bytes that can be transferred per unit time, or data transfer rate of the disk. Effective data transfer rates for the CURRENT and FRONTIER systems were taken from Worthington et al and the Cheetah datasheet respectively. The trend was extrapolated to get the throughput for the FUTURE system.

In a dedicated log disk, writes are streamed to disk in a sequential fashion and no other activity interferes with this and moves the head out of position. The head writes a track completely and then moves on to the next track to write that track in a similar sequential fashion. Thus, there is no seek cost associated with a write. However, a track-to-track switch occurs on certain writes when the current track is full and the write has to be continued on the next track. This cost is amortized over the capacity of each track. The amortized cost per byte of data is $\frac{t}{s}$ (as defined in table 5.1). The C2247 and Cheetah datasheets provide the values for t and s in the CURRENT and FRONTIER systems and the trend between

them was extrapolated for the FUTURE system.

The effective byte flush cost of the disk log is therefore

$$b = \frac{1}{B} + \frac{t}{s}$$

5.5 Modeling the Network Flush

In monolithic operating systems, most of the message processing takes place within the kernel. Every send and receive causes a protection domain crossing (from user to kernel mode) and the message is also copied from the kernel to the user address space. Network protocols are built using a layered approach which, while excellent for modularity and design, incurs a lot of overhead when compared to a single end-to-end implementation. New approaches in operating systems require user processes to undergo kernel interaction only to set up certain shared memory regions so that communication can then proceed without kernel calls. Liedtke [L93, L95] describes an architecture and kernel design for improving network communication.

My model of CURRENT and FRONTIER network configurations is based on the work of Maeda and Bershad [MB93], in which they propose an approach to network protocols that results in improved performance by separating the application's interface to the network from the application's interface to the operating system. The CURRENT network model is based on their implementation of the networking support in UX, the CMU single server UNIX operating system. The FRONTIER system is based on their in-kernel implementation of the Mach 2.5 operating system.

von Eicken et al [VBB95] propose a network architecture that allows user level access to high speed communication devices. They remove the kernel completely from the communication path while still providing full protection. Druschel and Peterson [DP93] develop another method to allow user level processes to take advantage of the bandwidth afforded by modern network devices. My FUTURE system is based on and uses data from [VBB95] as that study is more recent.

I assumed a dedicated network connection between the primary and the backup. The network bandwidth, B, for the CURRENT system (10 Mbps) was chosen to reflect the network connections between servers in the current Thor implementation. The network bandwidths for the FRONTIER (100 Mbps) and FUTURE (1 Gbps) systems are based on commercial networking technology available today and expected to be available in a few years respectively.

The cost of the flush is modeled as composed of the following operations: sending the flush message from the primary to the backup, processing at the backup, sending of the acknowledgement from the backup to the primary.

5.5.1 Flush Message Cost

I assumed a simple model of the network when estimating the costs of sending the flush message from the primary to the backup. The costs are composed of three actions: costs incurred at the sender, wire time of the message, and costs incurred at the receiver.

The sending and receiving costs are assumed to consist of a fixed cost (independent of the size of the message) and a variable cost (linearly proportional to the size of the message). Let fs and fr be the fixed sending and receiving costs respectively. Let the variable sending cost be vs per byte and the variable receiving cost be vr per byte. Wire times are computed as the transmission time for the message given the network bandwidth, B. The cost, MC, of transmitting a message of size M across a network is

$$MC = (fs + fr) + M * (vs + vr) + \frac{M}{B}$$

5.5.2 Acknowledgement Message Cost

I assumed that the costs of sending and receiving messages described in the pervious subsection were identical across servers. Therefore, the only difference between the acknowledgement cost and the flush message cost is the change in the size of the message. The cost of sending the acknowledgement, AC, from the backup to the primary is

$$AC = (fs + fr) + A * (vs + vr) + \frac{A}{B}$$

where A is the size of the acknowledgement. The size of the acknowledgement is independent of the size of the flush message. The Thor system uses 32 byte acknowledgements and this was assumed for all three system configurations.

5.5.3 Processing at the Backup

The processing at the backup involves appending the records that arrive in the flush message to the backup log. The Thor implementation employs efficient use of shared memory to avoid copying of data and has a fixed cost, BP, independent of the size of the flush data. The same cost, suitably scaled, was assumed for all three system configurations.

5.5.4 NetLog Flush Model

The latency of the NetLog flush operation is composed of the fixed parts of the flush message, the cost of processing at the backup and the cost of sending the acknowledgement from the primary to the backup. The latency is therefore

$$L = (fs + fr) + BP + AC$$

where BP is the cost of processing at the backup.

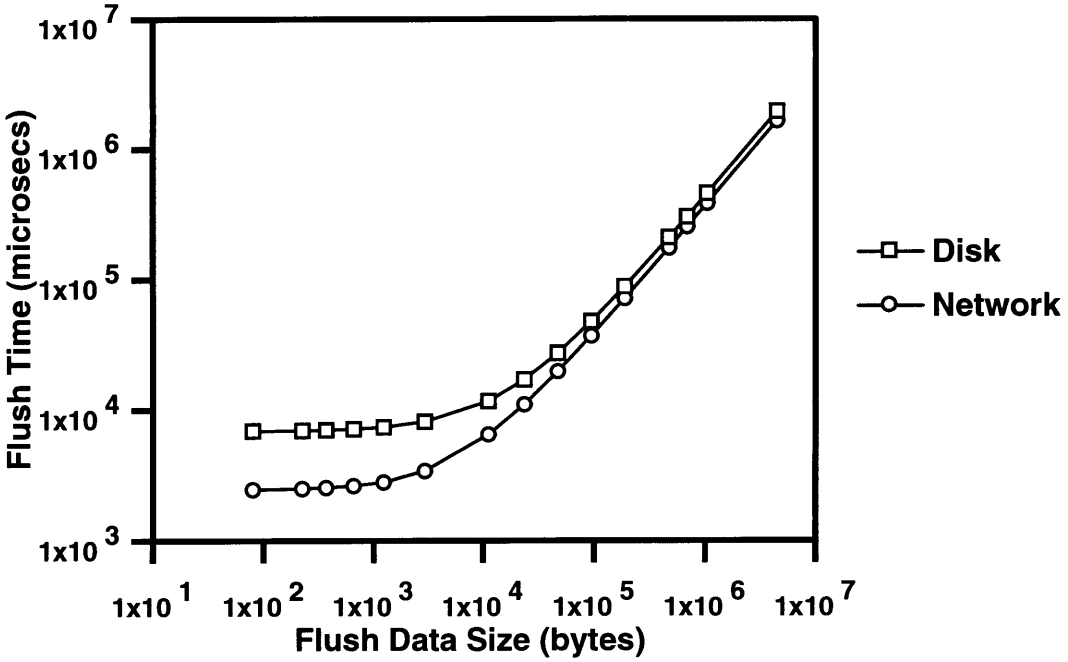

Figure 5-1: Variation of flush time with flush data size for the CURRENT system.

The byte flush cost of the NetLog flush operation is composed of the wire time for the flush message and the variable parts of the sending and receiving costs. The byte flush cost is therefore

$$b = (vs + vr) + \frac{1}{B}$$

The values of L and b were computed using the data from [MB93, VBB95].

5.6 Results

This section discusses the results obtained from the model.

5.6.1 Variation with Flush Data Size

Figure 5-1 shows the variation of flush times with increasing flush data sizes for the CURRENT system. A logarithmic scale was used for both axes. When comparing the NetLog and disk log, the NetLog is always faster than the disk log. The disk log has a high fixed cost and is 2-3 times slower than the NetLog for small flush data sizes (< 10 KB). As flush data size is increased, its performance relative to the NetLog gets better. After about 500 KB, it is seen from the graph that the two curves are almost parallel. In that range of

flush data sizes, the disk log is slower than the NetLog by about 10-20%. Extrapolating the model beyond the limits of the graph shows that the disk flush time is always greater than that of the NetLog but approaches it asymptotically.

Figure 5-2 shows the variation of flush times with increasing flush data size for the FRONTIER system. As before, the disk performance gets better as flush data size is increased. For small data sizes (< 10 KB), the disk flush latency is about an order of magnitude more than the network flush time. When the flush data size is between 500 KB to 5 MB, the disk log is about 10-30% slower than the NetLog. Extrapolating the model beyond the limits of the graph shows that, like the CURRENT system, the disk log is always slower than the network log but approaches the network log curve asymptotically.

For the FUTURE system, figure 5-3 shows the variation of flush times with flush data sizes. The behavior is very similar to the previous two graphs. For small data sizes (< 10 KB), the NetLog was about 30 times faster than the disk log. As flush data sizes are increased, the disk performance gets better. For flush data sizes between 500 KB to 5 MB, the disk log is about 25-35% slower than the NetLog. As in the previous two systems, the disk log is always slower than the network log regardless of flush data size and the disk curve approaches the network log curve asymptotically.

5.7 Variation with Time

Figure 5-4 shows the variation of the speedup of the NetLog over the disk log for all three system configurations. A logarithmic scale is used for speedup. The graph indicates that the NetLog performance relative to the disk log is expected to improve with time. The NetLog is thus better poised than the disk log to take advantage of technological developments in the coming years.

5.8 Number of Backups

So far, it has been assumed that the log records are sent to only one backup. If the probability of independent failure of any server is p, then in a three machine system, the probability that information is lost is the probability that two servers are down simultaneously, i.e. p^2. For example, if the downtime of systems today is assumed to be 5%, the probability of data loss on a three server system is therefore 2.5%. If higher reliability is desired, a simple solution is to send the log records to more than one backup. In a system with n backups, the probability of data loss is p^{n+1}. But flushing to more backups is costly as now n message transfers are required for each operation. If it is assumed that the primary sends the messages to the backups one after another. The presence of a broadcast medium will reduce the transmission time significantly, as the primary sends the flush message only once and waits for and processes n acknowledgements. In either case, sending data to a number of backups involves additional complexity and overhead in the flush process but provides greater fault tolerance.

I now extend the model to incorporate flushing to a number of backups. I assume the

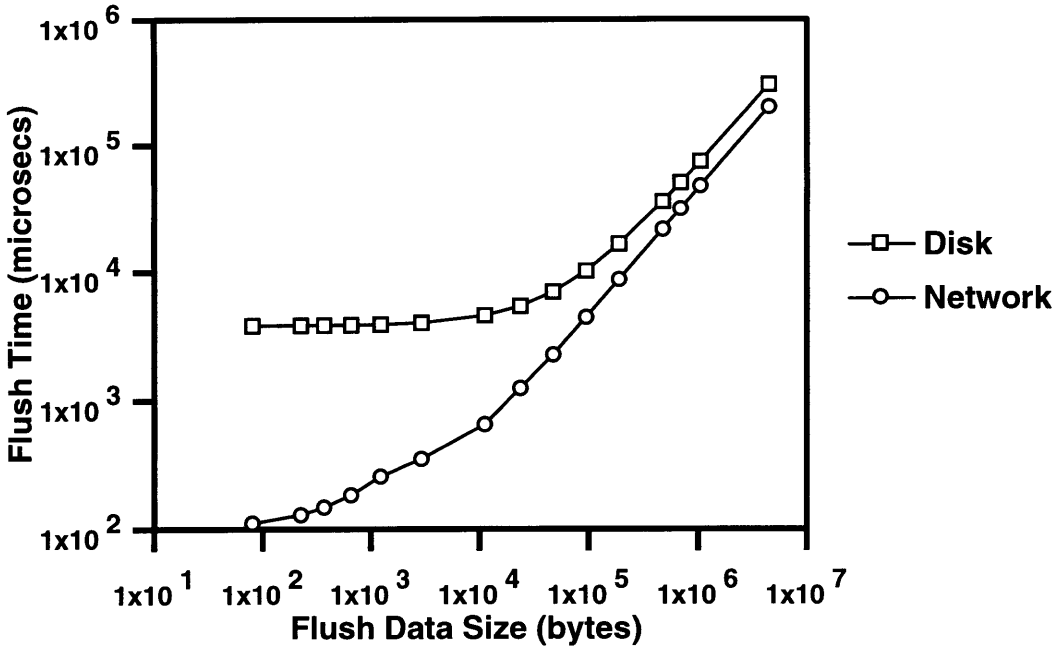

Figure 5-2: Variation of flush time with flush data size for the FRONTIER system.

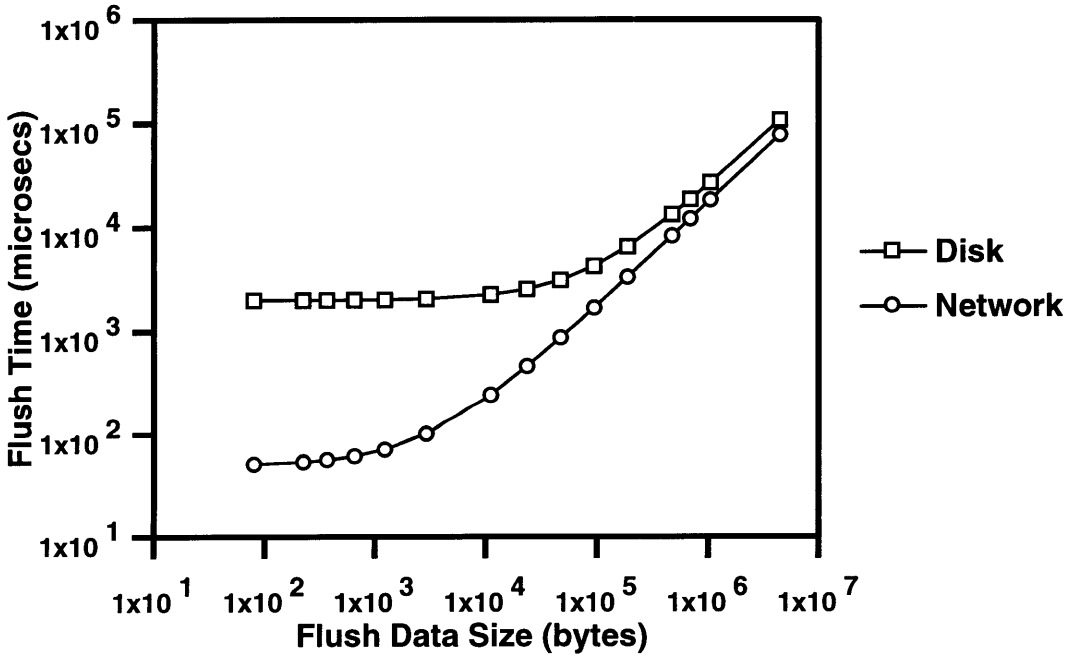

Figure 5-3: Variation of flush time with flush data size for the FUTURE system.

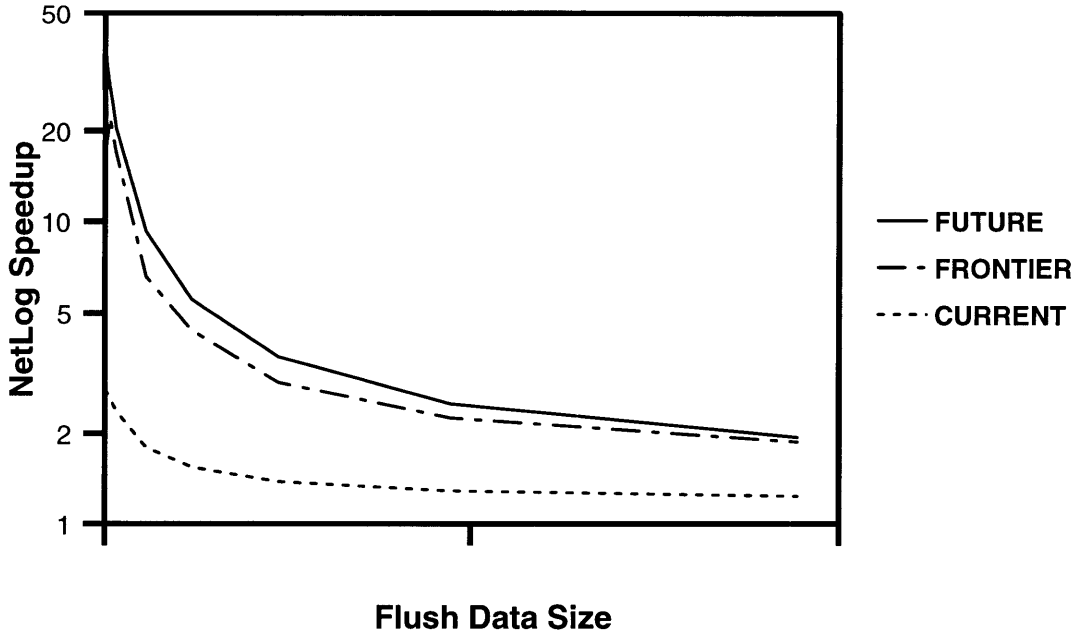

Figure 5-4: Variation of NetLog speedup over the disk log with time.

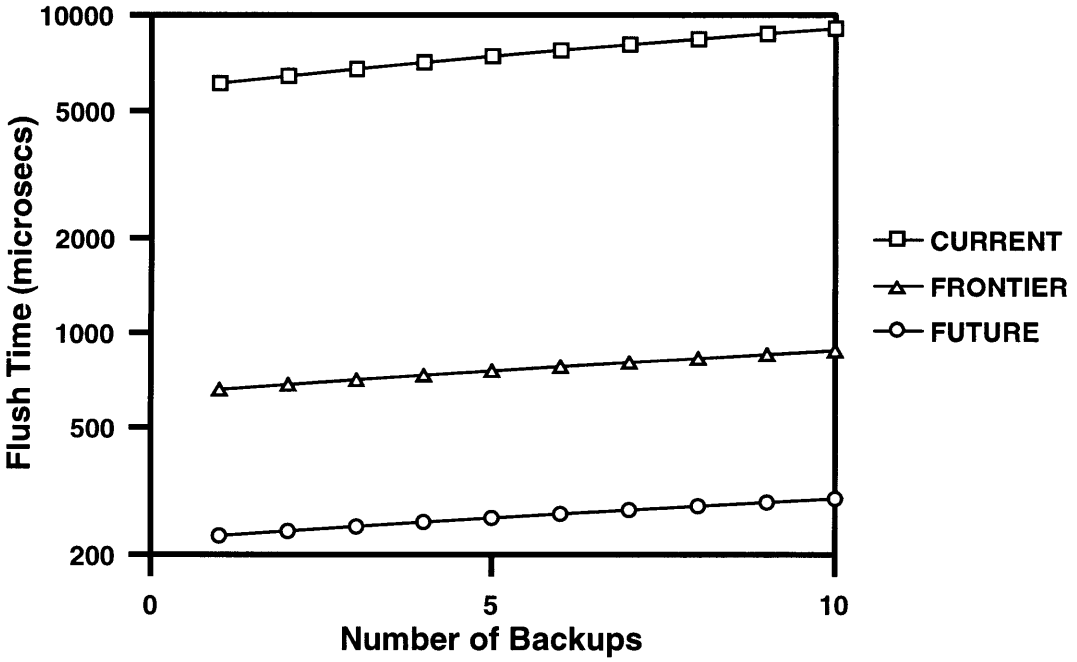

Figure 5-5: Variation of network flush time with number of backups.

presence of a broadcast medium on which the primary broadcasts flush data to the backups. Each backup then inserts the records into its memory and sends an acknowledgement to the primary.

The backups are numbered from 1 to n to organize the sending of acknowledgements to the primary. Backup i waits for the acknowledgement from backup $i-1$ before sending the acknowledgement to backup $i+1$. Backup 1 sends the acknowledgement to backup 2 after it has received the flush message from the primary and has appended the data to its log. On receiving the acknowledgement from backup $n-1$, backup n sends an acknowledgement to the primary. When this acknowledgement reaches the primary, the flush operation completes.

In this protocol, the cost of the flush operation consists of the cost of the flush message from the primary, processing at each of the backups (which is assumed to happen in parallel), and the cost of n acknowledgement messages. In terms of the model of section 5.3, the latency of the flush operation to n backups for a message of size M is

$$L = (fs + fr) + BP + n * AC$$

and the byte flush cost is

$$b = \frac{M}{B}$$

where B is the bandwidth of the network connection.

Using the data of the three settings, figure 5-5 shows the variation of the flush times with number of backups for both protocols and for the three different system settings. Though the graph assumes that the flush data size was fixed at 10 KB, similar behavior was observed for other data sizes. Note that the y axis is not drawn to scale. Over the three system settings, addition of another backup increases the flush time by about 10% on average. Note that this small increase is because of the assumption of a broadcast medium and ordered acknowledgements. In figure 4-1, as part of the implementation results, where a broadcast medium was not used, it was seen that the time for a two backup flush is slightly more than double the time for a one backup flush. The difference made by using a broadcast medium is now appreciated.

Note that the disk log flush times on the CURRENT, FRONTIER and FUTURE settings for 10 KB of data are 11000, 4500 and 2000 μsecs respectively; each of which is significantly more than the corresponding NetLog implementation that survives 10 server failures.

5.9 Conclusion

This chapter described the development and results of an analytical model that extended the results of the previous chapter to more general system configurations. The results of the model are consistent with the results of the Thor implementation and suggest that the NetLog performs as good or better than the disk log. From the model, it was seen that the performance of the NetLog relative to the disk log is expected to get better with advances

in computer technology. Increased reliability and availability may be obtained by flushing to more than one backup and the presence of a broadcast medium reduces the increased flushing cost associated with each additional backup significantly.

Chapter 6

Conclusion

This thesis has described the design, implementation, and performance of the NetLog, a distributed log abstraction that provides highly availlable stable storage. The performance of the NetLog was examined both as part of a real system as well as through analytical modeling. This chapter summarizes the work of this thesis and also presents some directions for future work.

6.1 Summary

There is a growing need among systems that store information on the Internet to provide highly available storage for data objects. The NetLog is a distributed log abstraction that provides servers with access to highly available stable storage through a simple, generic log interface. Since this interface is similar to that used by traditional log abstractions, servers can get the benefits of availability with little code change. However, the NetLog abstraction extends the traditional log abstraction by incorporating functionality required by certain applications, for example, the MOB in the Thor database. The NetLog uses the primary copy replication technique to provide highly available storage. All the details of the protocol are encapsulated within the log and hidden from servers. An important feature of the NetLog design is its independence from the nature or semantics of the objects stored within it.

The NetLog abstraction was implemented and used in the Thor distributed database to demonstrate its utility and study its performance in a real system. A disk log like that commonly used for stable storage in transaction systems was developed to provide a point for performance comparison. The flush times for both log abstractions were studied as a function of flush data size. For flush data sizes less than 10 KB, the NetLog was found to be an order of magnitude faster than the disk log. As the flush data size was increased, the disk got progressively better compared to the NetLog but never surpassed it. The Thor results indicate that the NetLog performs better or as well as the disk log for a large range of flush data sizes.

An analytical model of the NetLog was developed to allow a projection of the perfor-

mance implications of the NetLog on various kinds of hardware. By making use of detailed studies of disks and networks, a fine grained model of the network and disk was developed. The model was used to examine three different technology settings: CURRENT, the state of ubiquitous technology today; FRONTIER, commercially available technology today; and FUTURE, an estimate of the state of the technology available a couple of years down the line. Over all three system configurations, the NetLog was found to be as fast or faster than the disk log regardless of the flush data size. An important result from the model is that the relative performance improvement of the NetLog over the disk log is expected to get better as technology advances. The NetLog thus provides increased availability compared to the disk log, both today and in the future, without any decrease in performance.

6.2 Future Directions

In this section, I detail some of the research opportunities that arise from this thesis or as natural follow-ons to the work described here.

6.2.1 Actual Workloads

This work has clearly established that the NetLog performs as well or better than a disk log for a large range of flush data sizes chosen to model what is used in real systems. The next step is to compare the performance of both abstractions on a real database workload. A real workload will take into account not just the size of the flush data but also the number of transactions that flush a certain amount of data and their distribution. Having multiple clients execute workloads will also allow the measurement of not only the flush latency but also transaction throughput at the server. Doppelhammer et al [DHK97] describe the performance of the SAP system on the TPC benchmark and provide many insights on studying database performance in real situations.

6.2.2 Optimizations

There are a number of optimizations proposed in the literature to improve the performance of both log abstractions that are not incorporated in either the model or in the implementation. Currently, the committing of a transaction requires four message transfers, namely, client to primary, primary to backup, backup to primary, and primary to client. This may be reduced to three message transfers where the client sends the commit message to the primary and the backups, i.e. client to primary and backups, backups to primary, and primary to client. Some of the issues involved in this, especially the impact on concurrency control for multi-OR transactions, are discussed in [P97]. Architecture that provides weaker consistency [LLS90] between the primary and the backups may be implemented to improve performance at the risk of a greater number of client aborts. Other optimizations include changing the system configuration to provide better performance. These include using RAID storage [CLG94] and striping log data across disks to get higher bandwidth or using a high speed medium that permits the primary to broadcast flush messages to the

backups. Special hardware support for replication and efficient log maintenance through NVRAM [BAD92] is also possible.

6.2.3 Security

In the discussion in this thesis, it was assumed that the primary and backup failures follow the *fail-stop* approach [S90]. The design of efficient log abstractions that allow for more complex failure behavior like Byzantine failures, server corruption, and break-in are avenues for future research.

Bibliography

[A94] A. Adya, *Transaction Management for Mobile Objects using Optimistic Concurrency Control*, Masters Thesis, Technical Report MIT/LCS/TR-626, M.I.T. Laboratory for Computer Science, Cambridge, MA, 1994

[AD76] P. Alsberg and J. Day, *A Principle for Resilient Sharing of Distributed Resources*, In the Proceedings of International Conference on Software Engineering, October 1976

[BAD92] M. Baker, S. Asami, E. Deprit and J. Ousterhout, *Non-Volatile Memory for Fast, Reliable File Systems*, In the Proceedings of ACM International Conference on Architecture Support for Programming Languages and Operating Systems, October 1992

[B78] J. Barlett, *A Non-Stop Operating System*, In the Proceedings of International Conference on System Sciences, January 1978

[B81] J. Barlett, *A Non-Stop kernel*, In the Proceedings of Symposium on Operating System Principles, December 1981

[BHG87] P. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, MA, 1987

[BEM90] A. Bhide, E. Elnozahy and S. Morgan, *Implicit Replication in a Network File Server*, In the Proceedings of the IEEE Workshop on the Management of Replicated Data, November 1990

[BJW87] A. Birrell, M. Jones and E. Wobber, *A Simple and Efficient Implementation for Small Databases*, In the Proceedings of ACM Symposium on Operating System Principles, December 1987

[BBG83] A. Borg, J. Baumbach and S. Glazer, *A Message System Supporting Fault Tolerance*, In the Proceedings of ACM Symposium on Operating System Principles, October 1983

[CDN93] M. Carey, D. DeWitt and J. Naughton, *The OO7 Benchmark*, In the Proceedings of ACM International Conference on Management of Data, May 1993

[CFZ94] M. Carey, M. Franklin and M. Zaharioudakis, *Fine-grained Sharing in a page server OODBMS*, In the Proceedings of ACM International Conference on Management of Data, May 1994

[CAL97] M. Castro, A. Adya, B. Liskov and A. Myers, *HAC: Hybrid Adaptive Caching for Distributed Storage Systems*, In the Proceedings of ACM Symposium on Operating System Principles, October 1997

[CLG94] P. Chen, E. Lee, G. Gibson, R. Katz and D. Patterson, *RAID: High-Performance, Reliable Secondary Storage*, ACM Computing Surveys, 26(2), pp. 145-188, June 1994

[CD95] D. Cheriton and K. Duda, *Logged Virtual Memory*, In the Proceedings of ACM Symposium on Operating System Principles, December 1995

[DST87] D. Daniels, A. Spector and A. Thompson *Distributed Logging for Transaction Processing*, In the Proceedings of ACM International Conference on Management of Data, May 1987

[D95] M. Day, *Client Cache Management in a Distributed Object Database*, PhD Thesis, M.I.T. Laboratory for Computer Science, Cambridge, MA, 1995

[DGS85] S. Davidson, H. Garcia-Molina and D. Skeen, *Consistency in a Partitioned Network*, ACM Computing Surveys, 17(3), pp. 341-370, September 1985

[DKO84] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker and D. Wood, *Implementation Techniques for Main Memory Database Systems*, In the Proceedings of ACM International Conference on Management of Data, 1984

[DHK97] J. Doppelhammer, T. Hoppler, A. Kemper and D. Koffman, *Database Performance in the Real World- TPC-D and SAP R/3.* In the Proceedings of ACM International Conference on Management of Data, May 1997

[DP93] P. Druschel and L. Peterson, *FBufs: A High Bandwidth Cross Domain Transfer Facility*, In the Proceedings of ACM Symposium on Operating System Principles, December 1993

[ES83] D. Eager and K. Sevcik, *Achieving Robustness in Distributed Database Systems*, ACM Transactions on Database Systems, 8(3), pp. 354-381, 1983

[ESC85] A. El-Abbadi, D. Skeen and F. Cristian, *Fault Tolerant Protocol for Replicated Data Management*, In the Proceedings of ACM Symposium on Principles of Database Systems, 1985

[ET86] A. El-Abbadi and S. Toueg, *Maintaining Availability in Partitioned Replicated Databases*, In the Proceedings of ACM Symposium on Principles of Database Systems, 1986

[FC87] R. Finlayson and D. Cheriton, *Log Files: An Extended File Service Exploiting Write Once Storage*, In the Proceedings of the ACM Symposium on Operating System Principles, November 1987

[G79] D. Gifford, *Weighted Voting for Replicated Data*, In the Proceedings of the ACM Symposium on Operating System Principles, December 1979

[G83] D. Gifford, *Information Storage in a Decentralized Computer System*, Technical Report CSL-81-8, Xerox Corporation, March 1983

[G78] J. Gray, *Notes on Database Operating Systems*, Technical Report RJ2188, IBM Research Laboratory, San Jose, CA, 1978

[GHO96] J. Gray, P. Hilland, P. O'Neil and D. Shasha, *The Dangers of Replication and and Solution*, In the Proceedings of ACM International Conference on Management of Data, May 1996

[G95] S. Ghemawat, *The Modified Object Buffer : A Storage Management Technique for Object Oriented Databases*, PhD Thesis, Technical Report MIT/LCS/TR-666, M.I.T. Laboratory for Computer Science, Cambridge, MA, 1995

[G97] R. Gruber, *Optimism vs. Locking : A Study of Concurrency Control for Client-Server Object-Oriented Databases*, PhD Thesis, Technical Report MIT/LCS/TR-708, M.I.T. Laboratory for Computer Science, Cambridge, MA, 1997

[H86] R. Hagmann, *A Crash Recovery Scheme for Memory Resident Data Systems*, IEEE Transactions on Computers, 35(9), September 1986

[H87] R. Hagmann, *Reimplementing the Cedar File System USing Logging and Group Commit*, In the Proceedings of ACM Symposium on Operating System Principles, November 1987

[HP92] The Hewlett-Packard Company, *HP C2240 Series 3.5-Inch SCSI-2 Disk Drive: Technical Reference Manual*, part number 5960-8346, 2^{nd} edition, April 1992

[HBJ90] A. Hisgen, A. Birrell, C. Jerian, T. Mann, M. Schroeder and G. Swart, *Granularity and Semantic Level of replication in the Echo Distributed File System*, In the Proceedings of the IEEE Workshop on Management of Replicated Data, November 1990

[KLA90] M. Kazar, B. Leverett, O. Anderson, V. Apostolides, B. Bottos, S. Chutani, C. Everhart, W. Mason, S. Tu, and E. Zayas, *Decorum File System Architectural Overview*, In the Proceedings of the Summer USENIX Conference, 1990

[LLS90] R. Ladin, B. Liskov, L. Shrira, *Lazy Replication: Exploiting the Semantics of Distributed Services*, In the Proceedings of the ACM Symposium on Principles of Distributed Computing, August 1990

[L93] J. Liedtke, *Architecture Design and Code for Kernel Design to Improve IPC*, In the Proceedings of ACM Symposium on Operating System Principles, December 1993

[L95] J. Liedtke, *On Micro Kernel Construction*, In the Proceedings of ACM Symposium on Operating System Principles, December 1995

[LGG91] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira and M. Williams, *Replication in the Harp File System*, In the Proceedings of ACM Symposium on Operating System Principles, October 1991

[LAC96] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, L. Shrira, *Safe and Efficient Sharing of Persistent Objects in Thor*, In the Proceedings of ACM International Conference on Management of Data, June 1996

[LC97] D. Lowell and P. Chen, *Free Transactions With Rio Vista*, In the Proceedings of ACM Symposium on Operating System Principles, December 1997

[L96] N. Lynch, *Distributed Algorithms*, Morgan Kaufmann, San Francisco, CA, 1996

[MB93] C. Maeda and B. Bershad, *Protocol Server Decomposition for High Performance Networking*, In the Proceedings of ACM Symposium on Operating System Principles, December 1993

[M97] U. Maheshwari, *Garbage Collection in a Large, Distributed Object Store*, PhD Thesis, Technical Report MIT/LCS/TR-727, M.I.T. Laboratory for Computer Science, Cambridge, MA, September 1997.

[O88] B. Oki, *Viewstamped Replication for Highly Available Distributed Systems*, PhD Thesis, Technical Report MIT/LCS/TR-423, M.I.T. Laboratory for Computer Science, Cambridge, MA, August 1988

[ONG93] J. O'Toole, C. Nettles and D. Gifford, *Concurrent Garbage Collection of a Persistent Heap*, In the Proceedings of the ACM Symposium on Operating System Principles, December 1993

[P86] J. Paris, *Voting with Witnesses: A consistency Scheme for Replicated Files*, In the Proceedings of IEEE International Conference on Distributed Computing Systems '86, 1986

[P97] A. Parthasarathi, *Replication during or after Validation: Issues in the Design of a Replication Scheme for a Distributed Database*, First MIT Student Symposium on Computer Systems, January 1997

[PG92] C. Polyzois and H. Garcia-Molina, *Evaluation of Remote Backups Algorithms for Transaction Processing Systems*, In the Proceedings of ACM International Conference on Management of Data, June 1992

[RW91] C. Ruemmler and J. Wilkes, *Disk Shuffling*, Technical Report HPL-91-156, Hewlett-Packard laboratories, Palo Alto, CA, October 1991

[RW93] C. Ruemmler and J. Wilkes, *Unix Disk Access Patterns*, In the Proceedings of USENIX '93, Sunset Beach, CA, January 1993

[RW94] C. Ruemmler and J. Wilkes, *An Introduction to Disk Drive Modeling*, IEEE Computer, 27(3), 17-28, March 1994

[SMK93] M. Satyanarayanan, H. Mashburn, P. Kumar, D. Steere and J. Kistler, *Lightweight Recoverable Virtual Memory*, In the Proceedings of the ACM Symposium on Operating System Principles, December 1993

[S97] Seagate Technology, *Innovative Engineering creates cool Cheetah Drive*, 1997

[S90] F. Schneider, *Implementing Fault Tolerant Service using the State Machine Approach : A Tutorial*, ACM Computing Surveys, 22(4), pp. 299-319, December 1990

[S92] Standard Performance Evaluation Corporation, *Spec Benchmark Suite Release 2.0*, 1992

[SBH93] G. Swart, A. Birrell, A. Hisgen, C. Jerian and T. Mann, *Availability in the Echo File System*, Research Report 112, Systems Research Center, Digital Equipment Corporation, September 1993

[TWL92] C. Thekkath, J. Wilkes and E. Lazowska, *Techniques for File System Simulation*, Technical Report HPL-92-131, Hewlett-Packard Laboratories, Palo Alto, CA, October 1992

[T79] R. Thomas, *A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases*, ACM Transactions on Database Systems, 4(2), pp. 180-209, June 1979

[VT88] R. van Renesse and A. Tanenbaum, *Voting with Ghosts*, In the Proceedings of IEEE International Conference on Distributed Systems, 1988

[VBB95] T. von Eiken, A. Basu, V. Bush, and W. Vogels, *U-Net: A User Level Network Interface for Parallel and Distributed Computing*, In the Proceedings of ACM Symposium on Operating System Principles, December 1995

[V56] J. von Neumann, *Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components*, in Atomata Studies, Princeton University Press, pp. 43-98, 1956

[WGP94] B. Worthington, G. Ganger, and Y. Patt, *Scheduling for Modern Disk Drives and non-Random Workloads*, Technical Report CSE-TR-194-94, University of Michigan Department of Electrical Engineering and Computer Science, Ann Arbor, MI, 1994